

EAGER EVALUATION OF TASKS
IN A WORKFLOW SYSTEM

5 Cross Reference to Related Applications

Sub A' 7 This application is related to U.S. patent application serial no. _____ (attorney docket no. Dong 1-4-3-1-3), entitled Declarative Workflow System Supporting Side-Effects; U.S. patent application serial no. _____ (attorney docket no. Dong 2-7-5-3-2-2-5), entitled Data Item Evaluation Based on the Combination of Multiple Factors; and
10 U.S. patent application serial no. _____ (attorney docket no. Hull 6-2-1), entitled Dynamic Display of Data Item Evaluation; all of which are being filed concurrently herewith.

Field of the Invention

15 The present invention relates generally to task scheduling in computer systems. More particularly, the present invention relates to the eager evaluation of tasks in a workflow system.

Background of the Invention

20 In order to improve the performance of computer systems which execute multiple tasks during processing, it is known to execute tasks in an eager fashion. That is, the computer system will use available computing resources to execute tasks prior to fully determining whether the execution of those tasks is required for processing. Such eager execution generally improves the performance of the system by reducing response time
25 (i.e., the time it takes to process inputs to the system) because computing resources are more fully utilized. Of course, one result of eager execution is that certain tasks will be executed unnecessarily because it may be eventually determined that an eagerly executed task was not necessary for processing. However, overall performance is generally improved by the technique of eager execution.

30 A workflow system processes objects in accordance with a workflow specification. Such objects may be, for example, incoming calls to a call center, insurance claims arriving at a claim center, requests for information from an Internet web

site, etc. The workflow specification describes how agents (humans) and/or components (software and hardware) interact to accomplish specific goals. Such components generally include components which are external to the workflow system.

5 Workflow systems of the type further described herein below execute tasks in order to process a received object. Therefore, such workflow systems would benefit from the eager execution of tasks. However, as described in further detail below, workflow systems which control external components in order to process objects introduce additional considerations into the determination of which tasks may be executed eagerly.

10 U.S. Patent No. 5,809,212 entitled Conditional Transition Networks and Computational Processes for Use Interactive Computer-Based Systems, is directed to a conditional transition network for representing a domain of knowledge in a computer system. That system utilizes eager execution to evaluate conditions which indicate whether certain tasks will be executed in order to improve its response time to requests for stored information. Although the system utilizes eager execution for condition
15 evaluation, it does not utilize eager execution of tasks. Further, the system is not a workflow system, and as such, the system does not control external components. Therefore, the system does not take into account the additional considerations resulting from the control of external components.

20

Summary of the Invention

We have recognized that performance of a workflow system, the specification of which satisfies certain design criteria, may be improved by the application of novel eager execution techniques. The specification of the workflow system includes the definition
25 of enabling conditions, each of which is associated with a particular task, and indicates whether the particular task is to be executed during processing of an incoming object. In addition, the execution of at least some of the tasks (so-called side-effect tasks) results in the initiation of a side-effect action performed by a component external to the workflow system.

30 In accordance with the invention, the determination of whether a task is to be eagerly executed or not is based, at least in part, on whether the task is a side-effect task

or a non-side-effect task. Since non-side-effect tasks have low processing overhead, a non-side-effect task may be eagerly executed even if it is not known whether its associated enabling condition will ultimately be true. The cost of the unnecessary processing of non-side-effect tasks is outweighed by the performance benefit of eager execution. Side-effect tasks, on the other hand, have high overhead associated with them. For example, a side-effect task may request some type of user input, or may initiate some external processing. As such, a side-effect task is eagerly executed only if it is known that its associated enabling condition will ultimately be true. This is an improvement over prior workflow systems which generally treat all tasks as side-effect tasks and wait until it is known that an associated condition will become true before executing the task. The distinction of side-effect and non-side-effect tasks in accordance with the principles of the invention allow for the exploitation of additional opportunities for eager execution.

In accordance with further aspects of the invention, the properties of eligible, unneeded, and necessary are determined for tasks to further improve the performance of the system. Tasks which are *eligible* for eager evaluation are tasks which may be immediately evaluated, but which may or may not be required for complete processing of the workflow for a given object. Tasks which are *unneeded* are tasks which are not needed for complete processing of the workflow for a given object. Tasks which are *necessary* are tasks which are known to be needed for complete processing of the workflow instance for a given object. By using these three characteristics of tasks, the performance of the workflow system can be improved. Tasks which are eligible but unneeded are not executed, and tasks which are eligible and necessary are given high priority because it is known that these tasks will be required for complete processing.

Novel algorithms for determining whether tasks are eligible, unneeded, or necessary are provided. In accordance with yet a further aspect of the invention, these algorithms use dependency graphs to determine these characteristics. Propagation rules are provided and the algorithms apply the propagation rules to propagate changes through the dependency graph.

These and other advantages of the invention will be apparent to those of ordinary skill in the art by reference to the following detailed description and the accompanying drawings.

Brief Description of the Drawings

5 Fig. 1 shows a high level block diagram of an object focused workflow system;

 Fig. 2 shows a graphical representation of an example declarative language specification;

 Fig. 3 shows a graphical representation of a flowchart module portion of the example declarative language specification;

10 Fig. 4 shows the textual representation of the flowchart module shown graphically in Fig. 3;

 Fig. 5 shows a graphical representation of a declarative module portion of the example declarative language specification;

15 Fig. 6 shows the textual representation of the declarative module shown graphically in Fig. 5;

 Fig. 7 shows the textual representation of a foreign module portion of the example declarative language specification;

 Fig. 8 shows the textual representation of a foreign module portion of the example declarative language specification;

20 Fig. 9 shows a graphical representation of a declarative module portion of the example declarative language specification;

 Fig. 10 shows the textual representation of the declarative module shown graphically in Fig. 9;

25 Fig. 11 shows the textual representation of a decision module portion of the example declarative language specification;

 Fig. 12 shows the textual representation of a foreign module portion of the example declarative language specification;

 Fig. 13 shows the textual representation of a foreign module portion of the example declarative language specification;

30 Fig. 14 shows the textual representation of a foreign module portion of the example declarative language specification;

Fig. 15 shows the textual representation of a decision module portion of the example declarative language specification;

Fig. 16 shows the textual representation of a decision module portion of the example declarative language specification;

5 Fig. 17 shows the textual representation of a decision module portion of the example declarative language specification;

Fig. 18 shows the textual representation of a decision module portion of the example declarative language specification;

10 Fig. 19 shows the textual representation of a decision module portion of the example declarative language specification;

Fig. 20 shows the textual representation of a foreign module portion of the example declarative language specification;

Fig. 21 shows the textual representation of a decision module portion of the example declarative language specification;

15 Fig. 22 shows the textual representation of a foreign module portion of the example declarative language specification;

Fig. 23 shows the textual representation of a decision module portion of the example declarative language specification;

20 Fig. 24 shows the textual representation of a decision module portion of the example declarative language specification;

Fig. 25 shows the textual representation of a decision module portion of the example declarative language specification;

Fig. 26 shows a data and enabling flow diagram;

Fig. 27 shows the typing rules for Combining Policy Language operators;

25 Fig. 28 shows a high level functional diagram of a decision engine;

Fig. 29 shows the finite state automata for a non-side-effect attribute;

Fig. 30 shows the finite state automata for a side-effect attribute;

Fig. 31 shows the condition tree for an example enabling condition;

Fig. 32 shows a data and enabling flow diagram;

30 Fig. 33 shows a dependency graph;

Figs. 34A - 34D show pseudo code for the basic algorithm for determining whether a task is eligible or unneeded;

Figs. 35A - 35G show pseudo code for the extended algorithm for computing whether an attribute is necessary;

5 Fig. 36 shows the finite state automata for decision modules;

Fig. 37 shows the finite state automata for computation rules;

Fig. 38 shows an illustrative display screen shot of the graphical user interface;

Fig. 39 shows an illustrative display screen shot of the graphical user interface;

and

10 Figs. 40A and 40B show pseudo code for the algorithm for maintaining and dynamically updating the graphical user interface display.

Detailed Description

15

A high level block diagram of an object-focused workflow system in which the present invention may be implemented is shown in Fig. 1. As used herein, a workflow system is a system that facilitates the systematic execution of activities and tasks on a class of similar input objects based on common modules and/or functions. The workflow system may treat most or all of the objects in a uniform way, or may treat the input objects in highly differentiated ways. An instance of the workflow includes the processing that is performed by the workflow system and by external components as a result of an input object being provided to the workflow system. Multiple instances of the workflow might execute simultaneously. The workflow system may operate in real-time, near real-time, or at slower speeds. In accordance with the embodiment shown in Fig. 1, the object-focused workflow system 100 implements a customer care system for processing incoming calls. Of course the techniques described herein may be used to implement many different types of systems, such as electronic commerce systems. The system 100 includes a Declarative Language (DL) specification 102, a decision engine 104, a graphical user interface (GUI) 105, and an abstraction layer 106.

30

DL specification 102 represents a specification describing the desired operation of the object-focused workflow system 100 upon the receipt of some object. For example,

the object may be the receipt of an incoming telephone call to the customer care system. At least a portion of the DL specification 102 includes a *declarative* description of the operation of the system. The declarative description explicitly describes the steps to be taken during workflow execution, but the order of those steps is implicit in the DL

5 specification 102. This is a departure from the prior art workflow systems which generally describe workflow *procedurally*, that is, the specification explicitly states the steps to be taken during workflow execution and the sequence of those steps. The benefit of a declarative workflow specification is that it allows a system designer to focus on the core logic of an application and lets the underlying computer system address
10 implementation details, including the particular order in which steps should be taken. The details of the DL specification 102 will be described in further detail below.

The decision engine 104 receives the DL specification 102 and processes objects in accordance with the DL specification 102. Such processing of objects includes the control of various external components 108 such as a private branch exchange (PBX)
15 110, interactive voice response unit (IVR) 112, outbound call processor 114, e-commerce server 116, and database 120. These components are external in that they are not part of the workflow system 100, but may be controlled by the workflow system 100 during processing of an object. Only one of each of these external components is shown in Fig. 1 for clarity, however external components 108 may include more than one of each of
20 these components. For example, it would be common for the external components to include multiple databases. Further, the external components 108 shown in Fig. 1 is only an illustrative list of the types of components that may be controlled by an object-focused system implementing a customer care system. An object-focused system implementing other types of systems would likely contain other types of components.

25 A graphical user interface 105 is provided for displaying a representation of the execution of the workflow on a computer display screen and will be described in further detail below.

An abstraction layer 106 is provided between the decision engine 104 and the external components 108 so that the decision engine 104 may communicate with and
30 control the external components using a high level communication protocol, without the need to deal with the particular communication protocols of each of the different external

components. This is particularly useful because the external components 108 will generally be heterogeneous with different components being manufactured by different companies. The abstraction layer 106 includes an integrated view 122, system wrappers 124, and data source wrappers 126. The system wrappers 124 provide abstract interfaces to the external components 108, exposing properties relevant to the workflow application while hiding irrelevant details concerning particularities of the components' interfaces and aspects of the components' behavior. The data source wrappers 126 provide abstract interfaces to the databases and reporting features of the other external components 108, exposing relevant data and update capabilities. The integrated view 122 provides a unified interface to the functionalities supported by the external components 108, using abstract data types or similar interfacing modules. For example, the integrated view 122 may expose a functionality without exposing what components the functionality is implemented by, and expose a computed data set without showing what databases the raw data is residing on.

The decision engine 104, GUI 105, and abstraction layer 106 are high level representations of functions that may be performed by one or more appropriately programmed computers. The components of such computers are well known in the art and the details of such computers will not be described herein.

The contents of the DL specification 102 will now be described. In the embodiment described herein, the DL specification 102 comprises a textual specification which is provided to the decision engine 104 for processing. However, the DL specification 102 may be represented in other ways as well. For example, the DL specification 102 may comprise a graphical specification which is provided to the decision engine via data structures. The description of the DL specification 102 will be provided in conjunction with an example application of a travel agency customer care system. It is noted that the sole purpose of describing the DL specification 102 in conjunction with a particular example is to provide a context for describing the principles of the present invention. The use of the travel agency example used herein provides such a context. The purpose of this description is not to provide a complete description of how to implement a travel agency call center. Therefore, the example graphical and textual descriptions included herein will only be described in enough detail so as to clearly

describe aspects of the present invention. Further, although a fair level of detail is provided with respect to the example, not all details of the figures (in particular the textual specification) will be described herein. Further, the example is not to be taken as a complete description of the specification needed to implement such a travel agency call center. Portions of the specification which would be required to implement the example have been omitted for clarity. Further, portions of the specification have been included solely for the purpose of describing a certain aspect of the invention, and such portions may be inconsistent with other portions of the specification which have been included to describe certain other aspects of the invention.

In accordance with the example, it is assumed that a travel agency call center receives incoming calls from customers. The object passing through the system in this example is an incoming call, and the system operates to determine how to route the incoming call and what so-called side-effects should be executed. Side-effects will be described in further detail below. Assume that the call center has multiple agents, and each agent has one of 4 skill levels numbered 1-4. Each of the skill levels represents a particular level of expertise in a different type of call. During operation, it is desirable to route calls to appropriate agents in order to best utilize the agents' skills. Further details regarding the example application will be described below.

Fig. 2 shows a graphical representation of a DL specification for one portion of the travel agency example. More particularly, Fig. 2 shows a graphical representation of the Routing_To_Skill module 202 which will determine, based on various parameters of an incoming call, how to route the call. The term module is used herein to describe a logical grouping of related processing functions. The input parameters to the Routing_To_Skill module 202 are ANI, DNIS, WEB_DB_LOAD, and PROMOS_OF_THE_DAY. The output parameters of the Routing_To_Skill module 202 are SKILL, ON_QUEUE_PROMO, and WRAP_UP. These input parameters and output parameters are called attributes. The term attribute is used herein to describe a data item associated with an object which may be evaluated during processing of the object. An attribute in an object-focused workflow system is similar to a variable in a procedural system.

For purposes of this discussion, since the Routing_To_Skill module 202 is the highest level module described, it is assumed that values for the input attributes to the Routing_To_Skill module are provided to the system prior to execution of the workflow. Such attributes which are provided from some external source are called *source*

5 *attributes*. Similarly, the output attributes of the Routing_To_Skill module are called *target attributes*, because these are the attributes for which values are produced when processing is completed. Attributes which are used during processing which are neither source attributes nor target attributes are called *internal* attributes.

There are four types of modules in a DL specification. *Declarative* modules have
10 a declarative semantics and contain sub-modules. Declarative modules are represented in the figures using rounded corner rectangles. *Decision* modules produce a single attribute value through a novel technique for aggregating and synthesizing information using computation rules and a combining policy which will be described in further detail below. Decision modules are represented in the figures using solid line hexagons.
15 *Flowchart* modules are specified using a flowchart language to describe the processing. Flowchart modules are represented in the figures using rectangles with a cut corner. *Foreign* modules are implemented using some type of external function (e.g. database query or web server routine). Foreign modules are represented in the figures with a rectangle. Each of these types of modules will be described in further detail below.

20 As shown in fig. 2, the Routing_To_Skill module 202 is made up of six sub-modules: Identify_Caller 210, Info_About_Customer 220, Info_From_Web 230, Promo_Selection 240, Routing_Decisions 250, and Calculate_Wrap_Up 260. Module 210 is represented as a rectangle with a cut corner indicating that it is a flowchart module. Modules 230 and 240 are represented as rectangles which indicates that these modules
25 are foreign modules and obtain values for attributes by executing external functions. Modules 220 and 250 are represented as rounded corner rectangles, which indicates that these modules are declarative modules which are specified at a lower level using another DL specification. These types of modules are used to conveniently represent at a high level a substantial amount of lower level processing. Such high level representation
30 makes it easier for a designer to visualize the overall function of a DL specification. Module 260 is represented as a hexagon, which indicates that this module is a decision

module and evaluates a single attribute using computation rules and a combining policy as will be described in further detail below.

Each module has input attributes and output attributes. As will become clear from the description which follows, the workflow system 100 is attribute-based, that is, much of the processing which occurs in the workflow system is based on the evaluation of attribute values. Attributes are assigned triples as follows: (State, Value, Exception-Value) where:

	State	\in	{VALUE, EXCEPTION, DISABLED, UNINITIALIZED, FAIL}
10	Value	=	a value if State = VALUE \perp otherwise
	Exception-Value	=	exception value if State=EXCEPTION \perp otherwise

The meaning of these states is as follows. VALUE indicates that the attribute currently has a value assigned to it. EXCEPTION indicates that there has been some error in evaluating the attribute. DISABLED indicates that it has been determined that the attribute will not be assigned any value. It is noted that a DISABLED State does not indicate any error, but is part of normal processing. UNINITIALIZED indicates that essentially no information is available about the attribute. FAIL indicates that the module defining the attribute was enabled (as described below), but aborted before producing a state of VALUE, EXCEPTION, or DISABLED for the attribute. It is also noted that attribute assignments are monotonic, that is, once an attribute is assigned a state other than UNINITIALIZED, further processing of the workflow object will not change the state and any assigned value.

The value of an attribute will be some value assigned to the attribute if the state of the attribute is VALUE. Otherwise, the value will contain \perp , which indicates no value. The Exception-Value of an attribute will contain an exception value indicating a particular error condition if the state of the attribute is EXCEPTION. Otherwise, the Exception-Value will contain \perp .

Modules also have associated states. Module states are UNINITIALIZED, SUCCESS, EXCEPTION, and DISABLED. The module state UNINITIALIZED indicates that essentially no information is available about the module. The module state SUCCESS indicates that the enabling condition (as described below) for the module was true, and the module executed successfully. The module state EXCEPTION indicates that the module was enabled, but an exception occurred. The module state DISABLED indicates that the module's enabling condition is false.

Each of the modules is associated with an enabling condition, which is a condition which determines whether the module will be evaluated for a given input object.

Enabling conditions can refer to attribute values, attribute exception values, attribute states (e.g., whether the attribute has a value or whether an exception occurred when attempting to evaluate it), module states and module exception values. The enabling conditions are graphically represented as broken line hexagons 211, 221, 231, 241, 251, 261. Enabling conditions 211, 251, and 261 contain TRUE, which will always evaluate to a true condition, and therefore the Identify_Caller module 210, Routing_Decisions module 250, and Calculate_Wrap_Up module 260 will be evaluated for each input object. Enabling condition 221 contains the expression: VAL (ACCOUNT_NUMBER). The function VAL (X) will return a true condition if the attribute X is in the state VALUE, otherwise, false will be returned. Therefore, the enabling condition 221 indicates that the Info-About_Customer module 220 will be evaluated if the attribute ACCOUNT_NUMBER is in the state VALUE. If the attribute ACCOUNT_NUMBER is in state EXCEPTION, DISABLED, or FAILED, then enabling condition 221 will evaluate to false and the Info_About_Customer module 220 will not be evaluated. If the attribute ACCOUNT_NUMBER is in state UNINITIALIZED, then enabling condition 221 cannot yet be evaluated. Thus, the evaluation of enabling condition 221 depends on the attribute ACCOUNT_NUMBER first receiving a state other than UNINITIALIZED. It is noted that this dependency is implicit in the DL specification and not explicitly specified by the system designer or programmer.

Various expressions can be used for enabling conditions. For example, enabling condition 231 indicates that the Info-From_Web module 230 will be evaluated if the value of the WEB_DB_LOAD attribute is less than 95% or if the ACCOUNT_NUMBER

attribute does not have a value. Enabling condition 241 indicates that the Promo_Selection module 240 will be evaluated if the CUST_REC.HATES-PROMOS attribute is false.

Fig. 2 gives a high level graphical representation of the DL specification of the Routing-To-Skill module 202. Relevant portions of each of the sub-modules will now be described. A graphical representation showing further detail of the Identify_Caller module 210 is shown in Fig. 3. The DL specification in textual language corresponding to the graphical representation of Fig. 3 is shown in Fig. 4.

Sub 137
 10 As shown in Fig. 4, the module name is specified in line 1, and an indication of which module the current module is a sub-module of is given in line 2. The next section defines the input attributes (line 3). The next section defines the output attributes (lines 4 – 14). Line 15 specifies the enabling condition, which corresponds to the enabling condition 211 shown in Fig. 2. The type of the module, in this case flowchart, is specified on line 16. The computation section of the textual specification (line 17) indicates how attributes are to be evaluated. For this module, the attributes will be evaluated according to the flowchart shown in Fig. 3. Of course, one skilled in the art could convert the flowchart of Fig. 3 to program code to implement the logic flow shown in Fig. 3. Such code is not included in Fig. 4 because it is not necessary for an understanding of the principles of the present invention. Finally, line 18 indicates that this module has a side-effect. The side-effect action is an IVR dip (line 19).

A side-effect action is an action which has a significant impact on a system or user that is external to the workflow system. Stated another way, the execution of a side-effect action imparts a substantial overhead on the workflow system. Some actions may be deemed as being side-effect actions because a cost is associated with each occurrence of the action. For example, queries against some databases may have no associated cost because the databases are maintained by the same organization that maintains the workflow system, while queries against other databases may have associated costs because the database is maintained by an external organization. An action may be considered to be a side-effect action if the effect of the action cannot be undone by a subsequent separate action. Side-effects may include actions such as executing financial transfers, issuing checks or other instruments of monetary value, invoking actions by

other workflow engines, updating databases that are used by other software systems or users, or engaging users in an activity. An internal action is an action that is not a side-effect action. As described above in connection with line 18 of Fig. 4, an indication of whether a module includes a side-effect action is included in the DL specification.

5 The details of how attributes are evaluated by the Identify_Caller module 210 are given in the flowchart of fig. 3. The decision of step 302 determines whether the attribute ANI has a value. As shown in Fig. 2, ANI is a source attribute which is an input to the high level Routing_To_Skill module 202, and represents the telephone number from which the incoming call originated. Since such a telephone number is not always
10 available, the decision in step 302 is needed. If the ANI attribute has a value, then in step 304 the ACCOUNT-NUMBER and CUST_REC attributes for the customer associated with the ANI are evaluated by performing a lookup to an external database. If the decision in step 306 indicates that one customer has been identified, then in step 308 the attributes ACCOUNT_NUMBER and CUST-REC are assigned the values retrieved in
15 step 304. The assigning of values to the attributes ACCOUNT_NUMBER and CUST-REC includes assigning values to the associated tuple: (State, Value, Exception-Value) for each of the attributes. Thus, the state of these attributes becomes VALUE, the value gets the value retrieved, and exception-value is assigned \perp , as described above. Further, in step 308, the attribute HOME_PHONE is disabled, such that its associated tuple (State, Value, Exception-Value) is updated such that state becomes DISABLED, value becomes
20 \perp , and exception-value becomes \perp . As described above, since attribute assignments are monotonic, the values and states for these attributes will not change during further processing of this particular incoming telephone call.

If the test in step 306 is false, then in step 310 the customer is asked for a home
25 phone number. Such a step may be performed by initiating such a request from an interactive voice response unit, such as unit 112 (Fig. 1). Step 310 specifies a side-effect action because it impacts the caller by asking him/her to input some information. Upon receipt of the home phone number, the ACCOUNT_NUMBER and CUST_REC attributes are retrieved in step 312 in a manner similar to that in step 304. If one
30 customer is identified then the test in step 314 will be true and in step 316, the ACCOUNT_NUMBER, CUST_REC, and HOME_PHONE attributes are assigned

values. If one customer is not identified, then the test in step 314 will be false and in step 318 the HOME-PHONE attribute is assigned, and the ACCOUNT_NUMBER and CUST-REC attributes are disabled.

Sub A47 The DL specification further defining the Info_About_Customer module 220 (Fig. 2) is shown graphically in Fig. 5 and textually in Fig. 6. This Info_About_Customer module 220 is a declarative module and is therefore further defined in terms of sub-modules. The Get_Recent_Contacts_For_This_Customer module 504, the Get_Recent_Purchases_For_This_Customer module 508, the Get_Account_History_For_This_Customer module 512, and the Calculate_Cust_Value module 528 will always be evaluated because their respective enabling conditions 502, 506, 510, 526 are always true. It is noted that the graphical representation of these modules indicate that they are foreign modules. Each of these modules performs an external database retrieval function. If the attribute RECENT_CONTACTS has a state of VALUE, then the enabling condition 514 will be True and the Calculate_Frustration_Score module 516 will be evaluated. If the attribute RECENT_CONTACTS has state EXCEPTION, DISABLED, or FAILED, then the enabling condition 514 will be False and the Calculate_Frustration_Score module 516 will not be evaluated. If the attribute RECENT_CONTACTS is in state UNINITIALIZED, then enabling condition 514 cannot yet be evaluated. Enabling conditions 518, 522 and 530 are evaluated in a similar manner. The modules 516, 520, 524, 528, and 532 are all represented as solid line hexagons, which indicates that these modules are decision modules and the processing of these modules is specified in terms of computation rules and a combining policy, as will be described in further detail below.

The corresponding textual DL specification of the Info_About_Customer module 220 is shown in Fig. 6. It is noted that the type of the module as specified in line 16 is declarative, indicating that the module is a high level abstraction of processing details which are specified using sub-modules with enabling conditions.

Returning now to Fig. 2, the Info_From_Web module 230 will now be described in further detail. Module 230 is represented as a rectangle, which indicates that this is a foreign module which obtains values for attributes by executing external functions. The enabling condition 231 of module 230 indicates that the module will only be evaluated if

the attribute WEB_DB_LOAD has a value which is less than 95% or if the attribute ACCOUNT_NUMBER has a state other than VALUE or UNINITIALIZED. The textual DL description of the Info_From_Web module 230 is shown in Fig. 7. The computation specified in line 13 indicates that data from an external web server will be obtained using the attributes ANI, HOME_PHONE, and ACCOUNT_NUMBER. The information returned will be assigned to the attribute WEB_DESTINATIONS, which will contain information regarding a customer's prior interactions with an associated Internet web site.

The Promo_Selection module 240 will now be described in further detail. Like module 230, module 240 is represented as a rectangle, which indicates that this is a foreign module which obtains values for attributes by executing external functions. The enabling condition 241 of module 240 indicates that the module will only be evaluated if the attribute CUST_REC.HATES_PROMOS has a value False. The textual DL description of the Promo_Selection module 240 is shown in Fig. 8. The computation specified in lines 15-18 indicates that data from an external source (e.g. a database, expert system, another workflow system) will be obtained using the input attributes. The information returned will be assigned to the attribute PROMO_HIT-LIST, which will contain a list of promotions which would be appropriate to present to a customer during a call.

The DL specification further defining the Routing_Decisions module 250 (fig. 2) is shown graphically in Fig. 9 and textually in Fig. 10. This Routing_Decisions module 250 is a declarative module and is therefore further defined in terms of sub-modules. The Ask_Reason_For_Call module 910 will be evaluated if the CUST_VALUE attribute has a value less than 7, as indicated by enabling condition 912. Module 910 is represented as a rectangle, which indicates that the module is a foreign module. This module 910 performs an IVR interaction asking the caller the reason for calling, and the reason is assigned to attribute IVR_CHOICE. Modules 920, 940, and 950 will all be evaluated because their associated enabling conditions 922, 942, and 952, are all True. Module 960 will not be evaluated if the attribute BUSINESS_VALUE is greater than 100 or if the attribute FRUSTRATION_SCORE is greater than 5. This enabling condition 962 illustrates that enabling conditions can also specify when a particular module is disabled, rather than specifying when it is enabled. Module 930 will be evaluated if the test in its

enabling condition 932 is True. The modules 920, 940, 950, and 960 are represented as hexagons, which indicates that these modules are decision modules and that their attributes are evaluated using computation rules and a combining policy. These modules will be described in further detail below. Module 930 is represented as a rectangle which indicates that this is a foreign module.

The corresponding textual DL specification of the Routing_Decisions module 250 is shown in Fig. 10. The type of the module (line 20) is declarative, and is therefore further defined in terms of sub-modules. Thus, the module is a high level abstraction of processing details which are specified using sub-modules with enabling conditions. It is noted that line 21 of the textual DL specification indicates that the module has a side-effect. This side effect is a result of the Calculate_Send_Bonus_Check module 930 and will be described below in conjunction with that module.

Referring back to Fig. 2, the final module in the Routing_To_Skill module 202 is the Calculate_Wrap_Up module 260. Calculate_Wrap_Up module 260 is graphically represented in Fig. 2 as a hexagon, which indicates that the module is a decision module and that the processing of the module is specified in terms of computation rules and a combining policy. The use of computation rules and a combining policy to evaluate attributes will now be described in detail.

In general, the format of computation rules and a combining policy within a DL specification is as follows:

Computation rules:

If condition-1 then Attribute \leftarrow term-1

If condition-2 then Attribute \leftarrow term-2

.

.

.

If condition-n then Attribute \leftarrow term-n

Combining Policy:

Program in Combining Policy Language (CPL) or CPL function.

In the above format, the specification of :

If condition then Attribute \leftarrow term

indicates that the term *contributes* to Attribute if the condition has a True value. Thus, each of the computation rules is evaluated to determine whether the condition is True or False. If the condition is True, then the term contributes to the attribute. If the condition is False, then the term does not contribute to the attribute.

- 5 After each of the computation rules has been evaluated, the attribute is assigned a value based on the combining policy language (CPL) program, or the CPL function (where the CPL function is specified by a CPL program). Thus, the computation rules only contribute their terms to attributes, which is different from assigning a value to the attribute. The attribute is only assigned a value based on the combining policy (e.g. CPL
- 10 program or CPL function). For example, a CPL function may indicate that the highest value of the contributed terms is to be assigned to the attribute or that the sum of all the contributed terms is to be assigned to the attribute.

- It is noted that the use of computation rules and a combining policy has been described in the context of the object-focused workflow system. As such, computation
- 15 rules and combining policies are used to assign values to attributes. It is to be understood, however, that the use of computation rules and combining policies is not limited to use in an object-focused system. More generally, computation rules and combining policies may be used to evaluate any type of data item, whether that data item is an attribute in an object-focused system, a variable in a procedural system, or some
- 20 other type of data item.

- CPL provides a flexible and powerful mechanism that allows designers to specify how computation rules are to be combined in order to assign a final value to an attribute. CPL will first be specified formally using mathematical notation such that one skilled in the art of computer science could implement the language. Following the formal
- 25 description, examples of how the language would be used to build useful combining functions will be described in conjunction with the ongoing example.

First, the values and types of the CPL language will be described. Then, the operators that form an algebra for the language will be defined.

- CPL applies on homogeneous collections of data and is based on a type system that
- 30 defines the following *value types*. Each CPL type admits \perp (standing for “undefined”) as a specific occurrence. The type definitions are as

follows.

- atom (e.g. bool, int, float, string)
- $\langle a_1 : T_1, \dots, a_n : T_n \rangle$ is a tuple type, if each T_i is a value type.
- $[T]$ is a homogeneous list type, if T is a value type.
- 5 • $\{T\}$ is a homogeneous bag type, if T is a value type.
- AM: $T_1 \times \dots \times T_n \rightarrow T$ is an atomic mapping type if T_1, \dots, T_n and T are atom types. This type allows the definition of arbitrary mappings over atoms.

Values in CPL are defined as follows:

- 10 • \perp is a CPL value called *undefined*. \perp can be of any type. In others terms, \perp belongs to all the domains of all the CPL types.
- Any atom's value is a CPL value.
- 15 • Any finite tuple $(a_1:v_1, \dots, a_n:v_n)$ of CPL values is a CPL value if a_1, \dots, a_n are names for the fields in the tuple and v_1, \dots, v_n are CPL values.
- Any bag and any list of CPL values of a given type is a CPL value. $\{a_1, \dots, a_n\}$ is used to represent the bag value that contains the values a_1, \dots, a_n . $[a_1, \dots, a_n]$ is used as a shorthand to represent the list value that contains the values a_1, \dots, a_n and whose first element is a_1 , second element is a_2 and n^{th} element is a_n .
- 20 • Any arbitrary function defined over atoms is a CPL value. A function f of type AM: $T_1 \times \dots \times T_n \rightarrow T$ has for input types the atom types $T_1 \dots T_n$ and for return type the atom type T .
- 25 • Any arbitrary function defined over atoms is a CPL value. A function f of type AM: $T_1 \times \dots \times T_n \rightarrow T$ has for input types the atom types $T_1 \dots T_n$ and for return type the atom type T .

30 In CPL, variables are assigned types using type inference, as commonly found in functional programming languages such as ML. The inference rules used in CPL are detailed below.

A CPL program is a sequence of declarations of variables and/or functions:

```

i22    ::= ...
i23    ::= ...
35     :  ::=  :
```

where i22, i23, ... are variable or function identifiers. A variable declaration has the form:

$x ::= e$

where e is an expression of CPL. A variable declaration is well formed if the free variables used in e are previously defined in another variable declaration statement (no recursive variable declaration is allowed). A function declaration has the form:

$f ::= (x_1, x_2, \dots, x_n) \rightarrow e$

where e is an expression of CPL. A function declaration is well formed if

- $\forall i \in 1 \dots n, x_i$ is a variable, and
- the free variables used in e either are previously defined in a variable declaration statement or belong to $\{x_1, x_2, \dots, x_n\}$. (Recursively defined functions are not allowed).

The syntax of expressions depends on their types. Built-in operations ($=, \leq, \geq$) that are associated with the built-in atom types are allowed. Also allowed are some Boolean expressions such as e_1 and e_2 , e_1 or e_2 , $\neg e_1$ and a conditional if e then e_1 else e_2 where e is a Boolean expression and e_1, e_2 have the same type. Expressions can also be constructed using the operators defined below. Fig. 27 presents the typing rules for these operators. Names starting with v in fig. 27 represent variable names, names starting with e represent terms in the CPL calculus, and names starting with t represent types. The notation $\sigma \vdash e : t$ indicates that the term e is assigned the type t under the substitution σ . If a type equation is a fraction, the numerator is the premise while the denominator is the consequence.

The interpretation of the operators of CPL is now described. $I^*(e)$ represents the interpretation of an expression e . If e is a variable name or a function name, $I^*(e) = I(e)$ where the interpretation I associates variables to values and function names to functions of the appropriate type. (If e is a constant, $I(e) = (e)$).

The *value* operator performs as follows:

$$I^*(\text{value}(e)) = \text{if } I^*(e) = \perp \text{ then False else, True}$$

The *AMapply* operator performs as follows:

$$I^*(AMapply(f, e_1, \dots, e_n)) = I(f)(I^*(e_1), \dots, I^*(e_n))$$

Where f is any atomic mapping that applies on n atom values and returns an atom value.

Constructor operators are now described. A constructor operator is an operator which builds a composite object (e.g. tuple, list, bag) from other objects.

The operator *tupling* is defined as follows:

$$\begin{aligned} I^*(\text{tupling}(a_1 := e_1, \dots, a_n := e_n)) &= \langle a_1 : I^*(e_1), \dots, a_n : I^*(e_n) \rangle \\ I^*(\text{tupling}(\perp)) &= \perp \end{aligned}$$

The operator *bagging* is defined as follows:

$$\begin{aligned} I^*(\text{bagging}(e_1, \dots, e_n)) &= \{ I^*(e_1), \dots, I^*(e_n) \} \\ I^*(\text{bagging}(\perp)) &= \perp \end{aligned}$$

The operator *listing* is defined as follows:

$$\begin{aligned} I^*(\text{listing}(e_1, \dots, e_n)) &= [I^*(e_1), \dots, I^*(e_n)] \\ I^*(\text{listing}(\perp)) &= \perp \end{aligned}$$

As a shorthand, we use $\langle a_1 := e_1, \dots, a_n := e_n \rangle$, $\{e_1, \dots, e_n\}$ and $[e_1, \dots, e_n]$ to respectively note *tupling* ($a_1 := e_1, \dots, a_n := e_n$), *bagging* (e_1, \dots, e_n), and *listing* (e_1, \dots, e_n).

Deconstructor operators are now described. A deconstructor operator is an operator which extracts a component object from a composite object.

The *unitval* operator is defined as follows:

$$\begin{aligned} I^*(\text{unitval}(S)) &= \text{UNITVAL}(I^*(S)) \\ \text{UNITVAL}(\{\}) &= \text{UNITVAL}(\perp) = \perp \\ \text{UNITVAL}(\{a_1, \dots, a_n\}) &= a_1 \text{ if } n = 1 \end{aligned}$$

$$= \perp \text{ if } n \geq 2$$

The operator $Proj_{ai}$ is a parameterized operator. A parameterized operator is an operator which is defined using a template with parameters. Specific operators are formed from a parameterized operator by specifying a value for these parameters. The $Proj_{ai}$ operator is defined as follows:

$$I^*(proj_{a_i}(< a_1 : e_1 \cdots, a_n : e_n >)) = I^*(e_i)$$

$$I^*(proj_{a_i}(\perp)) = \perp$$

10

where $i \in \{1 \cdots n\}$.

The *getelt*_i operator returns the i^{th} element of a list. It is defined over lists as follows:

$$I^*(getelt_i(L)) = GETELT_i(I^*(L))$$

$$GETELT_i([a_1, \dots, a_n]) = a_i \text{ if } 1 \leq i \leq n$$

$$GETELT_i([a_1, \dots, a_n]) = \perp \text{ if } i > n$$

$$GETELT_i(\perp) = \perp$$

20 where i is a positive integer.

The *factor* operator is a binary operator that is defined over lists and bags as follows:

$$I^*(factor(S, Q)) = FACTOR(I^*(S), I^*(Q))$$

$$FACTOR([], b) = []$$

$$FACTOR(\perp, b) = \perp$$

$$FACTOR(\{\}, b) = \{\}$$

$$FACTOR([a_1, \dots, a_n], b) = [(a_1, b), \dots, (a_n, b)]$$

$$FACTOR(\{a_1, \dots, a_n\}, b) = \{(a_1, b), \dots, (a_n, b)\}$$

30

The *map* operator is an operator that is defined over lists and bags follows:

$$I^*(map(f)(S)) = I^*MAP(f, I^*(S))$$

$$I^*MAP(f, []) = []$$

$$MAP(f, \{\}) = \{\}$$

$$MAP(f, \perp) = I^*(f, (\perp))$$

$$MAP(f, [a_1, \dots, a_n]) = [I^*(f(a_1)), \dots, I^*(f(a_n))]$$

$$MAP(f, \{a_1, \dots, a_n\}) = \{I^*(f(a_1)), \dots, I^*(f(a_n))\}$$

35

Where f is any CPL function with only one input parameter.

The *collect* operator is recursively defined over lists and bags as follows:

$$\begin{aligned}
 5 \quad I^*(\text{Collect}(id_\theta, \theta)(S)) &= \text{COLLECT}^*(I^*(S), id_\theta, \theta) \\
 \\
 &\text{COLLECT}([], id_\theta, \theta) = id_\theta \\
 &\text{COLLECT}(\{\}, id_\theta, \theta) = id_\theta \\
 &\text{COLLECT}([a], id_\theta, \theta) = a \\
 10 \quad \text{COLLECT}(\{a\}, id_\theta, \theta) &= a \\
 &\text{COLLECT}([a_1, a_2, \dots, a_n], id_\theta, \theta) = I^*(\theta(a_1, \text{COLLECT}([a_2, \dots, a_n], id_\theta, \theta))) \\
 &\text{COLLECT}(\{a_1, a_2, \dots, a_n\}, id_\theta, \theta) = I^*(\theta(a_1, \text{COLLECT}(\{a_2, \dots, a_n\}, id_\theta, \theta)))
 \end{aligned}$$

where θ is a collector. A collector is a complete binary operator with identity id_θ . A collector can be any function $T \times T \rightarrow T$ where T is any CPL type except an atomic mapping type. The table below gives the predefined collectors that are used in the CPL.

θ	Type
\cup (Union)	$\{T\}$
@ (concat)	$[T]$
or	Boolean
and	Boolean
+	integer
-	integer
*	integer
<i>sup</i>	atom

20 The \cup collector computes the (bag) union of two bags (double are not eliminated). The @ collector does the concatenation of two lists ($@([a_1, \dots, a_n], [l, \dots, b_k]) = [a_1, \dots, a_n, l, \dots, b_k]$).

In practice, the user can define new collectors either (i) by providing built-in collectors associated with built-in atom types, or (ii) by constructing new collectors using

the CPL language. Indeed, the CPL language permits the user to declare any binary function to be used as a collector.

Constructed operators will now be defined. A constructed operator is an operator which is equivalent to a sequence of CPL statements. A constructed operator can always
5 be defined using basic operators. Constructed operators are used in CPL as a short-hand to represent sequences of CPL statements that are frequently used in CPL programs.

Select operator: The select operator is an operator that is parameterized by a boolean CPL function. It applies on lists or bags. The typing rules for select are:

10

$$\frac{\sigma \vdash cd:t_1 \rightarrow \{true, false\}, \sigma \vdash S_1: \{t_1\}}{\sigma \vdash \text{select}(cd)(S_1): \{t_1\}}$$

15

where t_1 is any CPL type. $\text{select}(cd)(S_1)$ returns a subset R of elements of S_1 . An element e_1 of S_1 is in R if $cd(e_1) = true$.

The $\text{select}(cd)$ operator (on bags) is equivalent to the following sequence of CPL statements:

20

$$\begin{aligned} \text{cond}(cd) &::= x \rightarrow \text{if } cd(x) \text{ then } \{x\} \text{ else } \{\} \\ \text{select}(cd) &::= S \rightarrow \text{collect}(\{\}, \cup)(\text{map}(\text{cond}(cd))(S)) \end{aligned}$$

The $\text{select}(cd)$ operator (on lists) is equivalent to the following sequence of CPL statements:

25

$$\begin{aligned} \text{cond}(cd) &::= x \rightarrow \text{if } cd(x) \text{ then } \{x\} \text{ else } [] \\ \text{select}(cd) &::= L \rightarrow \text{collect}([], @)(\text{map}(\text{cond}(cd))(L)) \end{aligned}$$

Join operator: The join operator is binary operator that is parameterized by a boolean CPL function. The typing rules for join are:

30

$$\frac{\sigma \vdash cd:t_1 \rightarrow \{true, false\}, \sigma \vdash S_1: \{t_1\}, \sigma \vdash S_2: \{t_2\}}{\sigma \vdash \text{join}(cd)(S_1, S_2): \{\langle f_a: t_1, s_a: t_2 \rangle\}}$$

$$\frac{\sigma \vdash cd:t_1 \rightarrow \{true, false\}, \sigma \vdash S_1:[t_1], \sigma \vdash S_2:[t_2]}{\sigma \vdash \text{join}(cd)(S_1, S_2) : \{\langle f_a:t_1, s_a:t_2 \rangle\}}$$

5

where t_1 and t_2 are any CPL type. $\text{join}(cd)(S_1, S_2)$ returns a set R of tuples of the form $\langle f_a:t_1, s_a:t_2 \rangle$. A tuple $\langle e_1, e_2 \rangle$ is in R if e_1 is in S_1 , e_2 is in S_2 and $cd(e_1, e_2) = \text{true}$.

$\text{Join}(cd)$ is equivalent to the following sequence of CPL statements:

10

$$\begin{aligned} \text{cond}(cd) &::= x \rightarrow cd(x.f_a, x.s_a) \\ \text{inner_loop}(cd) &::= y \rightarrow \text{select}(\text{cond}(cd))(\text{factor}(y.s_a, y.f_a)) \\ \text{Join}(cd) &::= S_1, S_2 \rightarrow \text{collect}(\{\}, \cup)(\text{map}(\text{inner_loop}(cd))(\text{factor}(S_2, S_1))) \end{aligned}$$

15

Trans operator The trans operator is a binary operator that is parameterized by CPL function. The typing rules for trans are:

20

$$\frac{\sigma \vdash f:t_1, t_2 \rightarrow t_3, \sigma \vdash S_1:\{t_1\}, \sigma \vdash e_2:t_2}{\sigma \vdash \text{trans}(f)(S_1, e_2) : \{t_3\}}$$

$$\frac{\sigma \vdash f:t_1, t_2 \rightarrow t_3, \sigma \vdash S_1:\{t_1\}, \sigma \vdash e_2:t_2}{\sigma \vdash \text{trans}(f)(S_1, e_2) : [t_3]}$$

25

$\text{trans}(f)$ is equivalent to the following sequence of CPL statements:

30

$$\begin{aligned} g(f) &::= x \rightarrow (s.f_a, x.s_a) \\ \text{Trans}(f) &::= S, y \rightarrow \text{map}(g(f))(\text{factor}(S, y)) \end{aligned}$$

Enum operator The enum operator associates each element of a list with its position in the list. The typing rule for trans is:

35

$$\frac{\sigma \vdash L_1:[t_1]}{\sigma \vdash \text{enum}(L_1) : (\langle pos:int, val:t_1 \rangle)}$$

It is defined as follows:

$$\begin{aligned}
 I^*(\text{enum}(e_1)) &= \text{ENUM}(I^*(e_1)) \\
 \text{ENUM}(a_1, \dots, a_k) &= [\langle pos : 1, val : a_1 \rangle, \dots, \langle pos : k, val : a_k \rangle] \\
 \text{ENUM}([]) &= [] \\
 \text{ENUM}(\perp) &= \perp
 \end{aligned}$$

The enum operator is equivalent to the following sequence of CPL statements:

$$\begin{aligned}
 10 \quad rev_concat &::= l_1, l_2 \rightarrow l_2 @ l_1 \\
 reverse &::= l_1 \rightarrow \text{collect}([], rev_concat)(\text{map}(\text{listing})(l_1)) \\
 init_enum &::= x \rightarrow [\langle pos : 1, val : x \rangle] \\
 merge_enum &::= l_1, L_2 \rightarrow [\langle pos : l_1 \# 0.pos + L_2 \# 0.pos, val : l_1 \# 0.val \rangle] @ L_2 \\
 15 \quad enum &::= L \rightarrow reverse(\text{collect}([], merge_enum)(\text{map}(init_enum)(reverse(L))))
 \end{aligned}$$

Dot operator The dot operator is a binary operator that combines two lists by associating elements with same position. The typing rule for dot is:

$$\frac{\sigma \vdash L_1 : [t_1], \sigma \vdash L_2 : [t_2]}{\sigma \vdash \text{dot}(L_1, L_2) : [\langle f_a : t_1, s_a : t_2 \rangle]}$$

It is defined as follows:

$$\begin{aligned}
 25 \quad I^*(\text{dot}(e_1, e_2)) &= \text{DOT}(I^*(e_1), I^*(e_2)) \\
 \text{if } (k < n) : \\
 \text{DOT}([a_1, \dots, a_k], [l, \dots, b_n]) &= [\langle f_a : a_1, s_a : l \rangle, \dots, \langle f_a : a_k, s_a : b_k \rangle, \langle f_a : \perp, s_a : b_{k+1} \rangle, \\
 &\quad \langle f_a : \perp, s_a : b_n \rangle] \\
 \text{if } (k > n) : \\
 30 \quad \text{DOT}([a_1, \dots, a_k], [l, \dots, b_n]) &= [\langle f_a : a_1, s_a : l \rangle, \dots, \langle f_a : a_n, s_a : b_n \rangle, \langle f_a : a_{n+1}, s_a : \perp \rangle, \\
 &\quad \langle f_a : a_k, s_a : \perp \rangle] \\
 \text{if } (k = n) : \\
 \text{DOT}([a_1, \dots, a_k], [l, \dots, m_n]) &= [\langle f_a : a_1, s_a : l \rangle, \dots, \langle f_a : a_n, s_a : b_n \rangle] \\
 35 \quad \text{DOT}([a_1, \dots, a_k], \perp) &= [\langle f_a : a_1, s_a : \perp \rangle, \langle f_a : a_k, s_a : \perp \rangle] \\
 \text{DOT}(\perp, [l, \dots, b_n]) &= [\langle f_a : \perp, s_a : l \rangle, \langle f_a : \perp, s_a : b_n \rangle], \\
 \text{DOT}(\perp, \perp) &= \perp
 \end{aligned}$$

The dot operator is equivalent to the following sequence of CPL statements:

```

5  to_one           ::= x → 1
   count           ::= L → collect(0,+)(map(to_one,x))
   choose_first    ::= x,y → x
   first           ::= L → collect(⊥,choose_first)(L)
   cd              ::= x → x.f.a.pos = x.s_a.pos
10  same_pos        ::= x → ⟨f_a:x.f_a.val;s_a :
   (first(select(cd)(factor(x.s_a,x.f_a))))f_a.val⟩
   rev_same_pos    ::= x → ⟨f_a: (first(select(cd)(factor(x.s_a,x.f_a))))f_a.val; s_a : x.f_a.val⟩
   dot             ::= L1,L2 → if count(L1) >= count(L2) then
15   map(same_pos)(factor(enum(L1), enum(L2)))
   else map(rev_same_pos)(factor(enum(L2),enum(L1)))

```

Group operator The group operator is a parameterized operator. The typing rules for group are:

```

20  
$$\frac{\sigma \vdash S: \{\langle hash:t'; val:t \rangle\}}{\sigma \vdash \text{group}(S): \{\langle hash:t'; vals:\{t'\} \rangle\}}$$

25  
$$\frac{\sigma \vdash S: \{\langle hash:t'; val:t \rangle\}}{\sigma \vdash \text{group}(S): \{\langle hash:t'; vals:\{t'\} \rangle\}}$$


```

where t' is an atom type.

Group takes as input a bag (resp, a list) and produces as output a bag of tuples of the form (h_i, S_i) $i \in 1 \cdots n$ such that

- $\forall i, j \in 1 \cdots n, (h_i \neq h_j) \text{ or } (i=j)$ and,
- Each member s_i of S_i contains the projection on the second attribute of all the tuples in S whose first attribute is equal to h_i .

The Group Operator is equivalent to the following sequence of CPL statements:

$\pi(hash) ::= S \rightarrow \text{map}(\text{proj}(hash))(S)$
 $\text{test_eq} ::= x \rightarrow \text{if } x.f.a = x.s_a \text{ then } true \text{ else } false$
 5 $I s_in ::= x, S \rightarrow \text{collect}(false, \text{or})(\text{map}(\text{test_eq})(\text{factor}(S, x)))$
 $\text{add_to_set} ::= s, S \rightarrow \text{if } I s_n(\text{unitval}(s), S) \text{ then } S \text{ else } s \cup S$
 $\text{elimin_double} ::= S \rightarrow \text{collect}(\{\}, \text{add_to_set})(\text{map}(\text{bagging})(S))$
 $\pi_{set}(hash) ::= S \rightarrow \text{elimin.double}(\pi(hash)(S))$
 $\text{same_hash} ::= x \rightarrow \text{if } x.f.a.hash = x.s_a \text{ then } \{x.f.a.val\} \text{ else } \{\}$
 10 $\text{all_same_hash} ::= y \rightarrow \{hash : y.f.a.vals : \text{collect}(\{\}, \cup)(\text{map}(\text{same_hash})(\text{factor}(y.s_a, y.f.a)))\}$
 $\text{Group} ::= S \rightarrow \text{collect}(\{\}, \cup)(\text{map}(\text{all_same_hash})(\text{factor}(\pi(hash)(S), S)))$

15 **Sort operator:** The sort operator is a parameterized operator. The typing rules for sort are :

$$\frac{\sigma \vdash \alpha : t \times t \rightarrow \{\text{True}, \text{False}\}, \sigma \vdash S : \{t\}}{\sigma \vdash \text{sort}(\alpha)(S) : [t]}$$

$$\frac{\sigma \vdash \alpha : t \times t \rightarrow \{\text{True}, \text{False}\}, \sigma \vdash S : [t]}{\sigma \vdash \text{sort}(\alpha)(S) : [t]}$$

25 where α is a CS mapping that describes an order relation over values of type T . More precisely, α is a CS mapping that takes as input two values of type T and returns a boolean value. α returns the value **True** if the first argument is greater than the second argument and **False** otherwise. α describes a non-strict order relation if it verifies the

30 properties (1) (2) and (3) described below. α describes a strict order relation if it verifies the properties (1) and (4).

- (1) if $(\alpha(x, y) \text{ and } \alpha(y, z))$ then $\alpha(x, z)$
- 35 (2) $\alpha(x, x) = \text{True}$
- (3) $\text{not}(\alpha(x, y)) \text{ or } (\text{not}(\alpha(y, x)) \text{ or } (y = x))$

(4) $\text{not}(\alpha(x,y))$ or $(\text{not}(\alpha(y,x)))$

The sort operator performs as follows: $\text{sort}(\alpha)(S)$ sorts the value of S in an increasing order according to α . Let us note that the result is not deterministic since (i) the order relation α is applied on a bag and (ii) α may define a partial order.

The sort operator is equivalent to the following sequence of CPL statements:

```

10       $f(\alpha)$                 ::=  $x \rightarrow \alpha(x.s\_a, x.f\_a)$ 
       $\text{select\_in } f$           ::=  $y, L \rightarrow \text{select}(f(\alpha))(\text{factor}(L, y))$ 
       $\text{select\_not\_in } f$       ::=  $y, L \rightarrow \text{select}(\text{not}(f(\alpha)))(\text{factor}(L, y))$ 
       $\text{merge\_order}$            ::=  $l, L \rightarrow \text{select\_in } f(l\#0, L) @ l @ \text{select\_not\_in } f(l\#0, L)$ 
       $\text{sort}(\alpha)$               ::=  $S \rightarrow \text{collect}([], \text{merge\_order})(\text{map}(\text{listing})(S))$ 

```

15 The foregoing was a formal definition of the CPL language. Examples of how the language is used to implement combining policy will be given below in connection with the ongoing example.

Returning now to the description of the example application and Fig. 2, module 260 is specified in terms of computation rules and a combining policy, and is shown textually in Fig. 11. The computation rules are specified in line 25 – 39 of the DL specification. As defined in lines 21-22, the output attribute WRAP_UP is a set of tuples containing an attribute name and a value of the attribute. In effect, this attribute contains various attribute names and associated values for archival purposes. The first three rules (lines 26-31) are of the form “if true”, so that they will always be True, and the values specified in those rules will always contribute to the attribute WRAP_UP. The computation rule on lines 32-35 will only contribute to the attribute WRAP_UP if the attribute WEB_DESTINATIONS is not empty. Similarly, the computation rule on lines 36-39 will only contribute to the attribute WRAP_UP if the value of CUST_REC.CARD_COLOR is “gold”. The combining policy (wrap-up-cp) is specified in lines 40-41 and indicates that the contribution of all true rules are to be used. It is assumed that this combining policy is a predefined CPL function which is available for use by system designers. The CPL program defining this function is as follows:

```
wrap_up_cp ::= x -> select(value)(x)
```

It is also noted that the Calculate_Wrap_Up module 260 has a side-effect, as specified in line 42. The side-effect of this module is to use the WRAP_UP attribute to write into an archive. This archive may be, for example, an external database which stores operational information for the system.

5 Returning now to Fig. 2, module 220 will be described in further detail. As described above in conjunction with Fig. 5, module 220 contains 8 sub-modules 504, 508, 512, 516, 520, 524, 528, 532. Modules 504, 508, and 512 are modules which perform database retrievals in order to assign values to attributes. The DL specification for the Get_Recent_Contacts_For_This_Customer module 504 is shown in Fig. 12.

10 Based on the input attribute ACCOUNT_NUMBER, the module will perform a database query as specified in lines 13-16 in order to evaluate and assign a value to the output attribute RECENT_CONTACTS. The DL specification for the Get_Recent_Purchases_For_This_Customer module 508 is shown in Fig. 13. Based on the input attribute ACCOUNT_NUMBER, the module will perform a database query as specified in lines 10-13 in order to evaluate and assign a value to the output attribute RECENT_PURCHASES. The DL specification for the Get_Account_History_For_This_Customer module 512 is shown in Fig. 14. Based on the input attribute ACCOUNT_NUMBER, the module will perform a database query as specified in lines 14-17 in order to evaluate and assign a value to the output attribute ACCOUNT_HISTORY.

20

 Returning now to Fig. 5, the Calculate_Frustration_Score module 516 is represented as a hexagon, which indicates that the module is a decision module and that the processing of the module is specified in terms of computation rules and a combining policy. The DL specification for this module is shown in Fig. 15. The computation rules are shown in part in lines 7-19. There would likely be additional computation rules, but such rules are not shown because they are not necessary for an understanding of the relevant portions of Fig. 15. The input attribute to this module is RECENT_CONTACTS, which is a list as defined in Fig. 12, lines 4-10. Thus, the computation rule specified in lines 8-13 of Fig. 15 specifies that if the attribute RECENT_CONTACTS is defined for list element [1] (i.e., there is at least one recent contact specified in the RECENT_CONTACTS attribute), then the expression in lines

25

30

10-13 is evaluated and the result is contributed to the FRUSTRATION_SCORE attribute. Similarly, the computation rule specified in lines 14-19 of Fig. 15 specifies that if the attribute RECENT_CONTACTS is defined for list element [2] (i.e., there are at least two recent contacts specified in the RECENT_CONTACTS attribute), then the expression in
 5 lines 16-19 is evaluated and the result is contributed to the FRUSTRATION_SCORE attribute. Additional computation rules would likely be included in an actual implementation, but are not shown here. It is noted that in this example, any or all of the computation rules may evaluate to True, in which case the attribute FRUSTRATION_SCORE gets contributions from each of the true rules.

10 The combining policy for the Calculate_Frustration_Score module 516 is a CPL function, frustration-score-cp, as specified in lines 21-24 and indicates that the contributions of the true rules will be added together and rounded up to the next integer, up to a maximum of 10. The result is assigned to the attribute FRUSTRATION_SCORE. The CPL program defining this function is as follows:

```

15 Calculate_Frustration_Score (
    sum_true  ::= x  ->  collect (select(value), 0 , + )
    min_of    ::= x, v  ->  if x > v then v else x
    Frustration_score_cp  ::= cont_vals -> min_of
    (round_up(sum_true(cont_vals,1)) , 10)

```

20

The Calculate_Net_Profit_Score module 520 (Fig. 5) is represented as a hexagon, which indicates that this module is a decision module and that the processing of the module is specified using computation rules and a combining policy. The DL specification for this module is shown in Fig. 16. The computation rules are shown in
 25 lines 10-32. The input attributes to this module are RECENT_CONTACTS, RECENT_PURCHASES, ACCOUNT_HISTORY, and CUST_REC. The computation rules specified in lines 10-32 specify the contributions to the attribute NET_PROFIT_SCORE based on the input attributes. The combining policy specified in lines 33-35 is a CPL function, net-profit-score-cp, and indicates that the contributions of
 30 the true rules will be added together. The resulting sum is assigned to the attribute Net_Profit_Score. The CPL program defining this function is as follows:

```

    Net_Profit_Score_cp ::= cont_vals -> sum_true(cont_vals)

```

The Calculate_Late_Payment_Score module 524 (Fig. 5) is specified using computation rules and a combining policy and the DL specification for this module is shown in Fig. 17. The computation rules specified in lines 7-19 specify the contributions to the LATE_PAYMENT_SCORE attribute based on the input attribute. The combining policy as specified in lines 20-22 is a CPL function, late-payment-score-cp, and indicates, that the contribution of a true rule will be assigned to the attribute LATE_PAYMENT_SCORE, or if there is no rule evaluating to a true value, then the LATE_PAYMENT_SCORE attribute will be assigned the default value of 0. It is noted that a constraint on this type of combining policy is that only one computation rule may evaluate to true. This constraint could be tested for during a pre-processing step to make sure that any possible evaluation will result in only one computation rule being true. The CPL program defining this function is as follows:

```

15      choose_first ::= x,y -> x
      Late_Payment_Score_cp ::= cont_vals ->
      collect(0,choose_first)(select(value)(cont_vals))

```

The Calculate_Cust_Value module 528 (Fig. 5) is specified using computation rules and a combining policy and the DL specification for this module is shown in Fig. 18. The computation rules specified in lines 9-19 specify the contributions to the CUST_VALUE attribute based on the input attributes. The combining policy as specified in lines 20-23 is a CPL function, calculate-cust-val-cp, and indicates that the contributions of rules with a true condition are added and rounded up, with a maximum of 100. The result is assigned to the attribute CUST_VALUE. If no rule evaluates to a true value, then the CUST_VALUE attribute is assigned the default value 0. The CPL program defining this function is as follows:

```

25      Calculate_cust_Value_cp ::= cont_vals ->
      min_of(round_up(sum_true(cont_vals)),100)

```

The Calculate_Marketing_vs_Collections module 532 (Fig. 5) is specified using computation rules and a combining policy and the DL specification for this module is shown in Fig. 19. The computation rule specified in lines 8-16 specifies the contributions to the MARKETING_VS_COLLECTIONS attribute based on the input attributes. It is noted that in an actual implementation additional rules would likely be included. The

combining policy as specified in lines 19-24 is a CPL function, marketing-vs-collection-cp, and indicates that the contribution of a true rule will be assigned to the attribute MARKETING_VS_COLLECTIONS, or if there is no rule evaluating to a true value, then the MARKETING_VS_COLLECTIONS attribute will be assigned the default value
 5 “marketing”. In this module, it is assumed that all computation rules (not all shown) have the effect of contributing the value "collect", and so there is no inconsistency if more than one rule has a true condition. The CPL program defining this function is as follows:

```
marketing_vs_collection_cp ::= cont_vals ->
10   collect(" marketing" ,choose_first)(select(value)(cont_vals))
```

Returning now to Fig. 2, module 250 will be described in further detail. As described above in conjunction with Fig. 9, module 250 contains 6 sub-modules. The Ask_Reason_For_Call module 910 will be evaluated if the CUST_VALUE attribute has
 15 a value less than 7, as indicated by enabling condition 912. Module 910 is represented as a rectangle, which indicates that the module is a foreign module. Module 910 requires an IVR interaction and asks the caller the reason for calling. The reason is assigned to attribute IVR_CHOICE. The DL textual specification for module 910 is show in Fig. 20. The computation steps are set forth in lines 8-12, which indicate that the attribute
 20 IVR_CHOICE will be assigned the value “dom” or “intl” or the state of this attribute will be EXCEPTION with an exception_value of 1, depending on the response to the IVR question. This module has a side-effect, as indicated in lines 14-15. The side effect is an IVR_dip, which will initiate an IVR question to the caller using the external IVR component 112 (Fig. 1).

25 Module 920 is represented as a hexagon, which indicates that this module is a decision module and is specified using computation rules and a combining policy. The DL specification for module 920 is shown in Fig. 21. The computation rules specified in lines 12-30 specify the contributions to the BUSINESS_VALUE_OF_CALL attribute based on the input attributes. The combining policy as specified in lines 31-34 is a CPL
 30 function, business-value-of-call-cp, and indicates that the contribution of the rules with a

true condition are added and rounded up with a default value of 0. The CPL program defining this function is as follows:

```

5      Business_Value_of_Call_cp ::= cont_vals ->
      round_up(sum_true(cont_vals))

```

The DL specification for the Calculate_Send_Bonus_Check module 930 is shown in Fig. 22. This module is represented as a rectangle, which indicates that this is a foreign module. This module will only be evaluated if the enabling condition specified in lines 5-13 is True. In such case, the module performs the side-effect action of issuing a check to a customer as specified in lines 16-17.

As shown in Fig. 9, all remaining sub-modules of the Routing_Decisions module 250 are represented as hexagons, which indicates that these modules are decision modules and are specified using computation rules and a combining policy. Turning now to the Calculate_Call_Priority module 940 module, the DL specification for this module is shown in Fig. 23. The computation rules specified in lines 8-20 specify the contributions to the CALL_PRIORITY attribute based on the input attributes. The combining policy as specified in lines 21-22 is a CPL function, call-priority-cp, and indicates that high value wins with a default value of 2. This means that the single highest contribution value for a true rule is the value that is assigned to the CALL_PRIORITY attribute. Thus, after all the computation rules are evaluated, there will be a collection of all the results, with the true rules contributing the value specified in the rule, and the false rules contributing \perp . The combining policy will choose the highest value in the collection and assign that value to the output attribute CALL_PRIORITY. If no rule has a true condition, then the value of 2 is assigned to CALL_PRIORITY. The CPL program defining this function is as follows:

```

30      choose_sup ::= x,y -> if (x >= y) then x else y;
      Call_Priority_cp ::= cont_vals -> collect(2<choose_sup)
      (select(value)(cont_vals))

```

The DL specification for the Calculate_Skill module 950 is shown in Fig. 24. The computation rules specified in lines 11-40 specify the contributions to the SKILL attribute based on the input attributes. The combining policy as specified in lines 41-45

is a CPL function, skill_cp, and indicates a weighted sum policy with ties being broken by the given ordering. Referring back to the computation rules, each true rule will contribute both a value and a weight to the SKILL attribute. For example, if the computation rule on lines 14-15 is evaluated to True, then the value high_tier with a weight of 40 is contributed to the SKILL attribute. After all the computation rules are evaluated, the sum of each of the weights for a particular value is computed, and the value that has the greatest weight is assigned to the SKILL attribute. If there is a tie between the weights of two different values, the value assigned to the SKILL attribute is determined by the ordering given by the combining policy. As an example, suppose the computation rule in line 28 and the computation rule in lines 32-35 are both evaluated to true. The computation rule in line 28 will contribute the value norm_tier_dom with a value of 30, and the computation rule in lines 32-35 will contribute the value norm_tier_dom with a value of 20. If these were the only two rules which contributed weights to the value norm_tier_dom, then the value norm_tier_dom would have a combined weight of 50 (30+20) when the combining policy was evaluated. If this combined weight of 50 for the value norm_tier_dom was the highest combined weight of any value, then the value norm_tier_dom would be assigned to the output attribute SKILL. The CPL program defining this function is as follows:

```

20      aggval ::= x -> tupling ( hash:= x.hash; val  :=
      sum_true(x,vals,0) )
      complist ::=
      [ " collections" , " australia_promo" , " high_tier" ,
        " low_tier_intl" , " low_tier_down" ]
      ccomplist ::= enum (complist)
25      compsel ::= x -> x.f_a.val = x.s_a
      poslist ::= x -> (select (compsel) (factor (ccomplist,x)) #0.f_a.pos
      compval ::= x,y -> if (x.val > y.val) then x
                          else if y.val > x.val then y
                          elseif poslist(x) < poslist(y) then x else y
30      Skill_cp ::= cont_vals -> collect(NULL, compval) (map (aggval)
      (group(cont_vals)))

```

The DL specification for the Calculate_On_Queue_Promo module 960 is shown in Fig.

25. The computation rules specified in lines 8-13 specify the contributions to the ON_QUEUE_PROMO attribute based on the input attribute. The combining policy as specified in lines 14-15 is a CPL function, on-queue-promo-cp, and indicates first true

wins with a default of 0. In accordance with this combining policy, the contribution of the true rule will be assigned to the attribute ON_QUEUE_PROMO. As described above, a constraint on this type of combining policy is that one and only one computation rule may evaluate to true. The CPL program defining this function is as follows:

```

5      on_Queue_Promo_cp ::= cont_vals -> collect (NULL,
          choose_first) (select (value) (cont_vals))

```

Returning to Fig. 1, the above description described the DL specification 102.

We now turn to a description of the decision engine 104 which will take the DL
 10 specification 102 and implement the specified workflow when an object (e.g., incoming call) is received at the workflow system 100. The following description will describe two embodiments of a decision engine 104. The first embodiment implements a straightforward execution of the workflow. The second embodiment implements an improved workflow execution which utilizes optimization strategies in accordance with
 15 aspects of the invention.

The first embodiment of the decision engine 104 requires that a topological sort of the modules be created. In order to describe the topological sort, first the data and enabling flow diagram shown in fig. 26 will be described. This diagram illustrates the data flow dependencies and the enabling flow dependencies of the workflow described
 20 above. Each of the modules (ovals) and enabling conditions (hexagons) are represented as nodes with solid line data flow edges representing data flow dependencies and broken line enabling flow edges representing enabling flow dependencies. Node 2601 represents the source attributes. A data flow edge from a module M to a module M' indicates that an output attribute of M is used as an input attribute of M'. An enabling flow edge from
 25 a module M to the enabling condition of a module M' indicates that the enabling condition of M' uses an output attribute from M. Also, there is an enabling flow edge from each enabling condition to the module that it qualifies. For example, there is a data flow edge 2602 from the identify_caller module 2604 to the info_about_customer module 2606 because the input attributes ACCOUNT_NUMBER and CUST_REC of the
 30 info_about_customer module 2606 are output attributes of the identify_caller module 2604. There is an enabling flow edge 2608 from the identify_caller module 2604 to the enabling condition 2610 of the info_about_customer module 2606 because the attribute

ACCOUNT_NUMBER used in the enabling condition 2610 is an output attribute of the identify_caller module 2604. There is an enabling flow edge 2612 from enabling condition 2610 to the module 2606 which it qualifies. Thus, fig. 26 shows the data flow and enabling flow dependencies for the routing_to_skill module 202 (Fig. 2). The data flow and enabling flow dependencies for lower level modules could similarly be shown using data and enabling flow diagrams. It is noted that the data and enabling flow diagrams of the modules are acyclic. That is, there is no module M for which there is a directed path in the graph composed of data flow and control flow edges that starts at M and ends at M.

The first step of the decision engine 104 is to produce a topological sort of the modules in the workflow description. As is well known, a topological sort is an ordering of modules such that the ordering satisfies the following properties:

If module M precedes module M' in the ordering then:

there is no directed path in the graph composed of data flow edges and enabling flow edges that starts at M' and ends at M.

Given the notation shown in Fig. 26, one topological sort for the modules shown in Fig. 26 is $M_1, M_2, M_3, M_4, M_5, M_6$. Topological sorts for lower level modules would be produced in a similar manner. After determining the topological sort, the modules may be executed such that a module is executed only after all preceding modules in the topological sort are completely finished executing and all attributes have been evaluated. Thus, given an ordering M_1, M_2, \dots, M_N , the modules are executed as follows:

enabling condition of M_1 is evaluated and if True, then module M_1 is completely executed, if False, then module M_1 is not executed;

enabling condition of M_2 is evaluated and if True, then module M_2 is completely executed, if False, then module M_2 is not executed;

.

enabling condition of M_N is evaluated and if True, then module M_N is completely executed, if False, then module M_N is not executed.

The above steps describe how a determination is made as to whether a module will be executed. With respect to how a module is executed, it depends on the type of module. If the module is other than a decision module and therefore specified in terms other than computation rules and a combining policy, then the module is executed as specified in the module (e.g. C++ program, database dip, flowchart, declarative module, etc.), and the details of such execution will not be described in detail herein. If the module is a decision module specified in terms of computation rules and a combining policy, then the module is executed as follows:

```

10      for each computation rule

        test the condition for True or False

          If False, produce  $\perp$  and add to collection

15          If True, then compute the value of the term on
            the right side of the rule and add to collection.

          (** at this point have a collection of values and/or  $\perp$  **)

20      apply combining policy program to the collection of values
        to produce the attribute value

        assign attribute value to the attribute;

25      execute any side-effect, if any.
```

The above described embodiment of the decision engine 104 executes the DL specification in a straightforward manner. That is, the various modules are executed sequentially in a topological order. However, the use of a more sophisticated execution technique can result in improved performance of the system. Such an execution technique will now be described.

For clarity of exposition, the more sophisticated execution technique for executing declarative modules will be described in a simplified context, but one skilled in the art will be able to extend the description presented here so that it can be used on arbitrary declarative modules. In the simplified context, the focus is on a DL program that consists of a declarative module with one or more internal modules. It is further

assumed that each internal module computes exactly one attribute, and may have a side-effect. It is further assumed that once enabled, internal modules will always produce a value and will never produce an exception value. Because each internal module produces only one attribute, a single state can be used for an attribute and the module that produces it. The states for attributes (modules) in this context are {UNINITIALIZED, VALUE, DISABLED}. Finally, suppose that module M produces attribute A . Because A is the only attribute produced by M , for convenience in exposition we refer to attribute A as a side-effect attribute if M is a side-effect module. Similarly, we refer to attribute A as a non-side-effect attribute if M is a non-side-effect module.

When describing the more sophisticated execution technique, the term task is used to refer to an execution of a module during execution of an instance of the workflow. Thus, a task refers to the activity (actual or potential) of executing a module. As will be described below, in some cases a task will be identified but not carried out. For example, a module and input values for it may be identified as eligible for execution but subsequently be deemed unneeded for completing the workflow and thus not executed. A task is a side-effect task if the module underlying the task is a side-effect module. A task is a non-side-effect task if the module underlying the task is a non-side-effect module.

A high level functional diagram of the decision engine 104 is shown in Fig. 28. The oval components represent data repositories and the rectangles represent software modules. The workflow schema 2810 represents the specification of how workflows are to be processed. For example, the workflow schema may be a DL specification as described above. Whenever a new object to be worked on is received (e.g. a new call enters a call center), a new instance of the workflow is created and stored in runtime workflow instances 2808. The execution engine 2812 works on a runtime workflow instance stored in 2808 in accordance with the workflow schema 2810 to execute the tasks in the workflow and to propagate the effects of the execution until the object has been fully processed. The execution engine 2812 works in a multi-thread fashion, so that it processes in parallel multiple workflow instances and multiple tasks within each instance. The task scheduler 2806 schedules the tasks that will be executed by the execution engine 2812. The task scheduler 2806 dynamically chooses the tasks to be

executed from the candidate task pool 2802, which is maintained by the prequalifier 2804.

The decision engine 104 executes tasks in an eager fashion, that is, the decision engine 104 will use available computing resources to execute tasks prior to fully determining whether such tasks are required for processing the workflow instance for a given object. Such speculative execution generally improves the performance of the system by reducing the response time (i.e., the time it takes to process inputs to the system) because computing resources will be fully utilized. Of course, by eager execution of tasks, certain tasks will be executed unnecessarily. However, the overall performance will be improved by eagerly executing certain tasks which, in fact, are needed to fully process the workflow instance.

The presence of side-effect modules imposes an important restriction on the eager evaluation of tasks. In particular, in a workflow instance a side-effect module should not be executed eagerly unless it is known that the module would also be executed in accordance with the declarative meaning of the DL specification. This restriction is motivated by the assumption that the processing impact of executing a side-effect module is so significant that the possible benefit of eager execution of the module is outweighed by the desire to avoid execution that is not justified by the meaning of the DL specification.

The prequalifier 2804 will determine three key properties of tasks which substantially improves the performance of the decision engine 104. Tasks which are *eligible* for eager evaluation are tasks which may be immediately evaluated, but which may or may not be required for complete processing of the workflow instance for a given object. It is noted that immediate execution of an eligible task will not violate the intended meaning of the DL specification. Tasks which are *unneeded* are tasks which are not needed for complete processing of the workflow instance for a given object. Tasks which are *necessary* are tasks which are known to be needed for complete processing of the workflow instance for a given object. By using these three characteristics of tasks, decision engine 104 can substantially improve its performance. Tasks which are eligible but unneeded may be deleted from the candidate task pool 2802, and tasks which are

eligible and necessary may be given high priority because it is known that these tasks will be required for complete processing.

Algorithms for determining whether tasks are eligible, unneeded, or necessary will be described below. Because of the complexity of the algorithms, the description will proceed in steps. First, an algorithm, identified as the *basic algorithm*, for
 5 determining whether tasks are eligible or unneeded will be described. Thereafter, an extension to the basic algorithm, identified as the *extended algorithm*, for further determining whether tasks are necessary will be described.

In order to describe the algorithms, several definitions must be introduced. For
 10 purposes of this description, assume that a workflow schema $S = (Att, Src, Tgt, Eff, Cnd, Mod)$ is fixed. This schema provides a mathematical formalism for describing DL specifications of declarative modules. The elements of the schema S are as follows:

Att is a set of attributes;
Src is the set of source attributes;
 15 *Tgt* is the set of target attributes;
Eff is the set of side-effect modules;
Cnd is the set of enabling conditions; and
Mod is the set of modules.

Recall that in the simplified context, each module produces as output a single
 20 non-source attribute. The algorithms described here focus on determining sets of attributes that are eligible, unneeded, or necessary. We also apply these terms to the modules associated with attributes, and to the tasks associated with those modules. For example, if attribute A is found to be eligible in the context of an execution of a
 25 workflow instance, then we also say that the module M that defines A is eligible in that context. Further, the task T of executing module M (whether this execution occurs or not) is said to be eligible in that context.

In order to implement the basic algorithm, additional states (i.e., in addition to those described above) for attributes must be defined. The states used in the algorithm
 30 are:

- UNINITIALIZED
- ENABLED
- READY
- READY+ENABLED
- 5 • COMPUTED
- VALUE
- DISABLED.

It is noted that the states EXCEPTION, and FAIL were defined above as attribute states,
 10 but are not used in conjunction with the simplified context for describing the execution algorithm.

In the context of the execution of a workflow instance, the states for an attribute A are defined as follows. Initially, an attribute A will be in the state UNINITIALIZED. This indicates that the attribute A has not yet been considered in the execution. The state
 15 ENABLED indicates that it has been determined that A 's enabling condition is, or will eventually be, True, but it is not yet determined whether A 's input attributes are stable (i.e., the input attributes are in the state "value" or "disabled"), and the value for A has not yet been computed. The READY state indicates that the input attributes for A are stable, but no determination has been made as to whether A 's enabling condition is true or false,
 20 and the value of A has not been computed. The state of READY+ENABLED indicates that the input attributes for A are stable and the enabling condition for A is true, but the value for A has not been computed. The state COMPUTED indicates that the value for A has been computed but it has not been determined whether the enabling condition for A is true or false. The state DISABLED indicates that the enabling condition for A is, or will
 25 eventually be, false. The state VALUE indicates that the value for A has been computed and the enabling condition for A is, or eventually will be, true.

Figs. 29 and 30 show the finite state automata (FSA) for this extended family of states. Fig. 29 shows the FSA for a non-side-effect attribute, and Fig. 30 shows the FSA for a side-effect attribute. The difference between the FSA for a non-side effect attribute
 30 (Fig. 29) and a side-effect attribute (Fig. 30) is that for side-effect attributes, the COMPUTED state is omitted. This is because, since the execution of modules computing side-effect attributes has significant impact on a system or user that is external to the workflow system, such modules should not be executed until it is known that their enabling conditions are, or eventually will be, true. For example, it would be undesirable

to initiate an IVR side-effect which asks a caller to provide certain information if that information is not required for complete processing of the object. The states VALUE and DISABLED are represented in Figs. 29 and 30 with double lines to indicate that they are terminal states for the attributes. If an attribute is in a terminal state then that attribute is

5 considered stable.

The notion of a *snapshot* will now be described. Snapshots are used to represent different stages in the execution of individual workflow instances. Let *Att* be a set of attributes with associated states. For each attribute *A* the type of *A* is denoted by $\tau(A)$. A *snapshot* for *Att* is a pair $s = (\sigma, \mu)$ where

- 10 1. the *state mapping* σ is a total function from *Att* into {UNINITIALIZED, ENABLED, READY, READY+ENABLED, COMPUTED, VALUE, DISABLED}
2. The *value mapping* μ is a partial function from *Att* into $\cup \{\tau(A) \mid A \in Att\}$, such that for each *A*, $\mu(A)$ is defined iff $\sigma(A) = \text{VALUE}$ or $\sigma(A) = \text{COMPUTED}$. If
- 15 $\mu(A)$ is defined then $\mu(A) \in \tau(A)$.

Thus, a snapshot is a data structure which stores the state (σ) of each attribute, and if the state of an attribute is VALUE then the data structure also stores the value (μ) of the attribute. The snapshot contains relevant information at a particular point during

20 workflow execution. As execution of the workflow progresses, the snapshot will change to reflect the new information determined during execution.

One snapshot s' *extends* another snapshot s (specified as $s < s'$) if for each attribute *A*:

- (a) if *A* has a value in s , then *A* has the same value in s' ; and
- 25 (b) the state of *A* in $s' \geq$ the state of *A* in s ;

where \geq is relative to the FSAs of Figs. 29 and 30. Thus, criteria (b) means that the state of *A* in s' is equal to, or at a higher level in the FSA than, the state of *A* in s .

One snapshot s' *strictly extends* another snapshot s if for at least one attribute *A*, the state of *A* in s' is $>$ the state of *A* in s .

- 30 A snapshot s is *complete* if each attribute *A* in s is stable. That is, each attribute has a state of VALUE or DISABLED.

For the purposes of describing the execution algorithm a particular formal definition for enabling conditions in DL specifications is now presented. One skilled in

the art will be able to use the techniques described here in connection with DL specifications that use a different formalism for enabling conditions and/or enabling conditions that can test values and properties different than those tested by the enabling conditions presented here.

- 5 A denotes attributes, $term$ denotes terms, and $pred$ denotes Boolean functions (such as comparison $term \theta term$) on terms which do not refer to states of attributes. The syntax of conditions is given by the following:

$$cond ::= pred(term, \dots, term) \mid VALUE(A) \mid DISABLED(A) \mid \neg cond \mid cond \wedge cond \mid cond \vee cond$$

- 10 The truth value of conditions is given by the standard two-valued logic when the involved attributes are stable, except that $pred(t_1, \dots, t_k)$ is true if all attributes referred to by $pred(t_1, \dots, t_k)$ have state VALUE and $pred(t_1, \dots, t_k)$ is true in the standard sense, and it is false otherwise. Thus, when evaluating an expression, if one or more of the attributes is in the state DISABLED, then the truth value is false. Note that this logic does not
- 15 always behave in the standard way (e.g. $A > 3 \vee A \leq 3$ is not a tautology). The *truth value* of a condition in a snapshot s where all attributes occurring in a condition γ are stable, is denoted $TruthVal(\gamma, s)$.

- A complete snapshot is *enabling-consistent* if for each attribute A which is not a source attribute, the state of A is VALUE if and only if the truth value of the enabling
- 20 condition of A relative to s is true.

- A second notion of consistency concerns the relationship between the values of the attributes and the modules that define them. To provide an interpretation for the behavior of modules we define an *environment* for schema S to be a mapping ε such that, for each module M in S_1 if M has input attributes I, \dots, B_n and output attribute A_1
- 25 then $\varepsilon(M)$ is a total mapping from $(T(B_1) \cup \perp) \times \dots \times (T(B_n) \cup \perp)$ to $T(A)$. The use of a static environment in this formalism is a convenience for this description. In practice, DL workflows will operate in the context of a dynamic world, i.e., the environment associated with a given workflow instance may be the result of combining the behaviors of different modules at different points in time, rather than from a single point in time.

- 30 A complete snapshot s is ε -consistent for environment ε if for each attribute A such that $\sigma(A) = \text{VALUE}$, $\mu(A)$ is equal to the A -value computed by $\varepsilon(M)(\mu(1), \dots,$

$\mu(B_n))$, where M is the module producing attribute A and has input attributes I, \dots, B_n .

(Note that if B_i does not have state VALUE in s for some i , then $\mu(B_i) = \perp$.)

An environment ε is compatible with snapshot s if it agrees with all defined values of μ in s .

- 5 For a snapshot s , $s|_{src}$ denotes the snapshot whose state and value functions agree with those of s on all source attributes, and such that all non-source attributes have state UNINITIALIZED.

- A snapshot s over S is *permissible* if (i) there exists an environment ε that is compatible with s , and (ii) for each environment ε compatible with s , if s' is a complete
10 snapshot that extends $s|_{src}$ and is compatible with ε , then s' extends s and the set of side-effect modules with state in $\{ENABLED, ENABLED+READY, VALUE\}$ in s is a subset of the set of side-effect modules with state VALUE in s' .

- It is noted that the notion of permissible snapshot captures in an absolute sense the family of snapshots s such that all determinations in s concerning whether attributes are
15 ENABLED or DISABLED and all side-effect actions that were executed in s are consistent with the declarative semantics associated with S . In practical situations (e.g., in situations where the condition language includes integers with addition and multiplication) the determination of whether a snapshot is permissible or not is undecidable, i.e., there is no algorithm that always terminates that determines whether,
20 for a given DL schema S and snapshot s , whether s is permissible for S . Even when the condition language is restricted to permit atomic types with equality, deciding whether a snapshot is permissible is intractable. However, it is possible to develop sufficient conditions for permissible that are tractable, even if the condition language is quite rich.

- In the algorithm developed here, execution of workflow S begins with a snapshot
25 such that all source attributes are stable and all other attributes are in state UNINITIALIZED. Then a sequence of permissible snapshots is constructed, each one a strict extension of the previous one. Execution halts when a terminal snapshot is reached.

A non-source attribute A is *absolute-enabled* for permissible snapshot s if for each complete snapshot s' that extends s , A has state VALUE. A non-source attribute A is

absolute-disabled for snapshot s if s is permissible and for each complete snapshot s' that extends s , A has state DISABLED.

Given a permissible snapshot s over S then:

- 5 • A side-effect attribute A is *absolute-eligible* in s if A is absolute-enabled and each input attribute for A is stable.
- A non-side-effect attribute A is *absolute-eligible* in s if each input attribute for A is stable.

A snapshot $s = (\sigma, \mu)$ over S is *terminal* if s is permissible and $\sigma(A) \in \{\text{VALUE}, \text{DISABLED}\}$ for each $A \in \text{Tgt}$. That is, a snapshot is terminal if it is
10 permissible and all target attributes are stable.

A snapshot s over S is *minimal terminal* if s is terminal and s is not a strict extension of another terminal snapshot for S .

An attribute A is *absolute-unneeded* for permissible snapshot s over S if for each minimal terminal snapshot $s' = (\sigma', \mu')$ that extends s , $\sigma'(A) \notin$
15 $\{\text{COMPUTED}, \text{VALUE}\}$.

As with the definition of permissible, the notions of absolute-eligible and absolute-unneeded define, in an absolute sense, all eligible attributes and all unneeded attributes, for a given permissible snapshot during execution of a workflow schema. However, the actual computation of all eligible or unneeded attributes is not possible in
20 practical situations, e.g., if the condition language includes integers with addition and multiplication. Even if the condition language is limited to include only atomic types with equality, computing all eligible or unneeded attributes is intractable. Thus, a subset of absolute-eligible and absolute-unneeded attributes is determined in order to improve the performance of a workflow execution.

25 The basic and extended algorithms are used to determine which attributes to evaluate eagerly, that is, which attributes should be computed even though not all of the attributes occurring in their associated enabling conditions have become stable. Such an analysis involves partial computations of conditions, since the conditions may depend on attributes which have not yet been computed. In order to represent such partial
30 computations, a three valued logic is used. The truth value for a given condition may be true, false, or unknown. Instead of considering each enabling condition as one indivisible

three-valued logic formula, enabling conditions are represented by trees. This gives more precise knowledge as to which sub-formulas are true and which are false. Condition trees are used for this purpose.

A *condition tree* of an enabling condition P is obtained from the parse tree (as well known in the art) of P by replacing each leaf node p of the form $pred(t_1, \dots, t_k)$ with a tree $T(p)$ defined as:

the root node of $T(p)$ is an AND operator node;

$pred(t_1, \dots, t_k)$ is a leaf node of $T(p)$; and

VALUE (A) is a leaf node of $T(p)$ if A occurs in $pred(t_1, \dots, t_k)$.

All the leaf nodes are directly connected to the root node.

For example, consider the enabling condition: (NOT(F=3 AND G=4)) OR DISABLED (F). The condition tree for this enabling condition is shown in Fig. 31.

Now, in order to determine which tasks may be computed eagerly, a dependency graph is defined which will take into account the dependencies between the enabling conditions and the attributes, and the dependencies between the attributes themselves. The dependency graph is defined as follows. Given a workflow schema S , the *dependency graph* of S , denoted D_S , is a directed acyclic graph that is constructed as follows:

- each enabling condition in the workflow schema S is represented by its condition tree in D_S ;
- each attribute in A is a node in D_S ;
- there is an edge from the root node of each condition tree to the attribute node attached to the associated enabling condition in S ;
- there is an edge from an attribute A to a predicate node p if and only if A occurs in p ;
- there is an edge from an attribute A to an attribute B if and only if A is an input attribute of B .

As an example, consider the workflow schema represented by the data and enabling flow diagram of Fig. 32. As described above in conjunction with the data and enabling flow diagram of Fig. 26, the data and enabling flow diagram of Fig. 32 illustrates the data flow

dependencies (solid lines) and the enabling flow dependencies (broken lines) for a given workflow schema. Given the workflow schema represented in Fig. 32, the dependency graph for that workflow schema is shown in Fig. 33. In Fig. 33, all nodes belonging to an evaluation tree are *condition nodes*, with the remaining nodes being *attribute nodes*. The edges between attribute nodes are shown with broken lines and are *data edges* and the edges between condition nodes are shown with solid lines and are called *condition edges*.

Finally, prior to describing the basic algorithm, the notion of an extended snapshot must be defined. An extended snapshot is a tuple

$s = (\sigma, \mu, \alpha, \text{Hidden-att}, \text{Hidden-edge})$ where:

- (a) the *state mapping* σ is a total function from *Att* into {UNINITIALIZED, ENABLED, READY, READY+ENABLED, COMPUTED, VALUE, DISABLED}
- (b) The *value mapping* μ is a partial function from *Att* into $\cup \{\tau(A) \mid A \in \text{Att}\}$, such that for each A , $\mu(A)$ is defined iff $\sigma(A) = \text{VALUE}$. If $\mu(A)$ is defined then $\mu(A) \in \tau(A)$
- (c) α maps each condition node to T (true), F (false), or U (unknown)
- (d) *Hidden-att* is the set of attributes which have been hidden (the notion of a hidden attribute will be described below); and
- (e) *Hidden-edge* is the set of edges (both data edges and condition edges) which have been hidden (the notion of a hidden edge will be described below).

The pseudo code for the basic algorithm for determining whether a task is eligible or unneeded in accordance with the invention is shown in Figs. 34A-D. Generally, the algorithm starts at the beginning of the execution of a workflow instance and ends when the execution is completed. The prequalifier 2804 (Fig. 28) executes this algorithm for each workflow instance. The algorithm computes and incrementally maintains the states (in an array $\sigma[]$) and the values (in the array $\mu[]$) of the attributes defined in the workflow schema 2810. In order to carry out its computations, the algorithm uses the dependency graph D_s . It incrementally computes a set of nodes called *hidden-att* such that the attribute nodes contained in the set *hidden-att* are stable or unneeded, and a set of edges called *hidden-edge* where edges contained in *hidden-edge* are edges which have been traversed by the algorithm, and do not have to be considered again by the algorithm. More generally, if an attribute or edge is "hidden", then the information embodied in that attribute or edge relevant to the algorithm has already been used during execution and

possibly incorporated into the snapshot, and will not be needed in any subsequent step of the algorithm. The algorithm also maintains an array of three valued logic values ($\alpha[]$) for condition nodes. The algorithm consists of two main phases. The first phase is an initialization phase which is executed once at the beginning of execution of a workflow instance. The second phase is the incremental phase which is executed each time a value of an attribute is computed by the execution engine 2812 and incorporated into the runtime workflow instances 2808. The incremental phase is also executed for the source attributes when the workflow instance is initially placed into workflow instances 2808.

The pseudo-code for the basic algorithm shown in Figs. 34A-D will now be described. Section 3402 of the algorithm is the definition of the global variables. Section 3404 is the definition of some of the notations used in the algorithm. The initialization phase 3406 is executed once at the beginning of execution of a workflow instance in order to initialize the required data structures. In section 3408, the states and values of the attribute nodes of the dependency graph are initialized such that the state of the source attribute nodes are initialized to a state of READY+ENABLED and all other attribute nodes are initialized to a state of UNINITIALIZED. The values (μ) of the attributes are set to null. In section 3410, the logic values ($\alpha[]$) for condition nodes are initialized to unknown. In section 3412, the set of hidden edges and hidden attributes is set to null. This is the end of the initialization phase 3406.

The remainder of the pseudo code defines the incremental phase. Each time a new value for an attribute is computed by the execution engine 2812 (Fig. 28), the increment function 3414 is called as part of the operation of the prequalifier 2804. The increment function is also called by the prequalifier 2804 at the beginning of processing a workflow instance, once for each source attribute. The increment function 3414 then calls the propagate_att_change function 3422 which in turns recursively calls the propagate_cond_change function 3450 in order to propagate changes along the dependency graph. Both the propagate_att_change function 3422 and the propagate_cond_change function 3450 call the recursively defined functions hide_edge 3474 and hide_node 3476. These functions will now be described in further detail.

The increment function is shown in section 3414. As shown by the input section 3416, this function is called when a value v has been computed for an attribute A in the

dependency graph G . First, the value (μ) of the attribute is updated in step 3418. Next, in section 3420 the `propagate_att_change` function is called. If the current state of attribute A is READY, then the `propagate_att_change` function is called with parameters indicating that the state of A should be updated to COMPUTED. If the current state of attribute A is ENABLED+READY then the `propagate_att_change` function is called with parameters indicating that the state of A should be updated to VALUE. The increment function then ends.

The `propagate_att_change` function is shown in section 3422. The input to this function, as shown in section 3424, is an attribute B and a state (σ) for B . In section 3426 the state (σ) for attribute B is updated. In section 3428, changes are propagated up the dependency graph as a result of the newly computed attribute as follows. If the state of the attribute B has changed to VALUE or COMPUTED, then in section 3430 an attempt is made to evaluate predicate nodes which use the value of B . Thus, for each condition node in which B occurs in the predicate (line 3432) and for which the edge in the dependency graph from B to the predicate is not currently hidden (line 3434), the edge is hidden (line 3436) and an attempt is made to evaluate the predicate using the new value of B . If the predicate can be evaluated, then the logic value (α) is updated to True or False and the change is propagated by calling the `propagate_cond_change` function (line 3438). The `propagate_cond_change` function will be described in further detail below. Thereafter, for each attribute node C which has B as an input attribute, if B has the state VALUE and the value for B is the last input attribute needed for C to go stable, then attribute C is promoted to the READY state by calling the `propagate_att_change` function (section 3440).

If the state of attribute B is ENABLED, then in section 3442 for each condition node p of the form $\text{VAL}(B)$ the state of p ($\alpha[p]$) is set to TRUE, and for each condition node p of the form $\text{DIS}(B)$ the state of p ($\alpha[p]$) is set to FALSE, and the changes are propagated by calling the `propagate_cond_change` function. This processing takes place because when it is known that an attribute B is enabled, then the truth value of $\text{VAL}(B)$ must be true, because the attribute will eventually be assigned some value. Similarly, the truth value of $\text{DIS}(B)$ is known to be false.

In a manner similar to that described above in conjunction with section 3442, if the state of attribute B is DISABLED, then in section 3444 for each condition node p of the form $VAL(B)$ the state of p ($\alpha[p]$) is set to FALSE, and for each condition node p of the form $DIS(B)$ the state of p ($\alpha[p]$) is set to TRUE, and the changes are propagated by

5 calling the `propagate_cond_change` function. Thereafter, in a manner similar to step 3440, for each attribute node C which has B as an input attribute, if the value for B is the last input attribute needed for C to go stable, then attribute C is promoted to the READY state by calling the `propagate_att_change` function (section 3446).

10 The last line 3448 of the `propagate_att_change` function indicates that if the state of B has become DISABLED or VALUE (i.e., attribute B has become stable), then the node is hidden.

The `propagate_cond_change` pseudo-code is shown in section 3450. This is a recursive algorithm which propagates condition changes up the dependency graph. The input to this function is a condition node p in the dependency graph G . The node n is the

15 parent of the node p (line 3452). If the edge from $p \rightarrow n$ is not hidden, then section 3454 is executed. Otherwise, the function ends. First, the edge from $p \rightarrow n$ is hidden (line 3456). If the parent of p (n) is an OR condition node section 3458 is executed. If the truth value of condition node p is true, then the truth value of the parent node n ($\alpha[n]$) is set to true (because an OR condition is true if any of its components is true) and the

20 change is further propagated up the dependency graph by recursively calling the `propagate_cond_change` function for node n (line 3460). If the truth value of condition node p is false and if there are no other non-hidden edges into the parent n , then the truth value of the parent node n ($\alpha[n]$) is set to false (because if there are no more non-hidden edges then there are no more possibilities for a component of n to be true), and the

25 change is further propagated up the dependency graph by recursively calling the `propagate_cond_change` function for node n (3462).

If the parent of p (n) is an AND condition node section 3464 is executed. If the truth value of condition node p is false, then the truth value of the parent node n ($\alpha[n]$) is set to false (because an AND condition is false if any of its components is false) and the

30 change is further propagated up the dependency graph by recursively calling the `propagate_cond_change` function for node n (line 3466). If the truth value of condition

node p is true and if there are no other non-hidden edges into the parent n , then the truth value of the parent node n ($\alpha[n]$) is set to true (because if there are no more non-hidden edges then there are no more possibilities for a component of n to be false), and the change is further propagated up the dependency graph by recursively calling the

5 `propagate_cond_change` function for node n (3468).

If the parent of p (n) is a NOT condition node then the truth value of the parent node n ($\alpha[n]$) is set to the inverse of the truth value of condition node p in 3470.

If the parent of p (n) is an attribute node, then the enabling condition for node n can now be evaluated and section 3472 of the pseudo code is executed. If the truth value of condition node p is true, then the `propagate_att_change` function is called to update the state of node n to ENABLED. Otherwise, if the truth value of condition node p is false, then the `propagate_att_change` function is called to update the state of node n to DISABLED.

The `hide_edge` function is shown in section 3474. The `hide_edge` function receives as input an edge (n, n') in g . The function will hide a node n if the received edge (n, n') is the last non-hidden edge emanating from n .

The `hide_node` function is shown in section 3476. The `hide_node` function receives as input a node n in g . The function hides all edges into n .

When the basic algorithm is finished executing, there is a new updated snapshot stored in the memory of the system. Because of properties of the algorithm, this snapshot is permissible. A determination as to which attributes are eligible is made as follows. A non-side effect attribute is eligible if its state (σ) is READY or READY+ENABLED. A side effect attribute is eligible if its state (σ) is READY+ENABLED. Recall that if an attribute A is determined to be eligible for execution then the task corresponding to execution of the module that defines A is also viewed as eligible for execution. Thus, in accordance with one aspect of the invention, since side-effect tasks have significant impact external to the workflow system, a side-effect task is eligible for eager evaluation only if the data is available for its evaluation and if it is known that its enabling condition will ultimately be true (i.e. the corresponding attribute is in state READY+ENABLED).

30 Non-side-effect tasks, on the other hand, have no significant external impact, and a non-side-effect task may be considered as eligible for eager evaluation when its data inputs

are available, whether or not it is known that its enabling condition will ultimately be true (i.e. the corresponding attribute is in state READY or READY+ENABLED).

Further, a determination as to which attributes, and hence which tasks, are unneeded is made as follows. An attribute A is unneeded if the node for A in the dependency graph is hidden and the state of A is not COMPUTED or VALUE. The node of an attribute may become hidden if its value will not be used, directly or indirectly, in the evaluation of any target attribute. As a particular example, if the attribute is an input for some target attribute but partial evaluation of the enabling condition for the target attribute indicates that the enabling condition will take the value FALSE, and the attribute will not be used, directly or indirectly in the evaluation of any other target attribute, then the node of the attribute will become hidden. Recall that if attribute A is unneeded then the task corresponding to execution of the module producing A is also viewed as unneeded.

At this point, the prequalifier 2804 (Fig. 28) identifies all tasks which are eligible and not unneeded as candidate tasks and provides these candidate tasks to the candidate task pool 2802. If a task which was previously identified as eligible is newly identified as unneeded, then the corresponding task is removed from the candidate task pool 2802.

In determining that an attribute is ENABLED, READY+ENABLED, or DISABLED the algorithm may, in accordance with one aspect of the invention, use a partial evaluation of the enabling condition of the attribute, i.e., an evaluation of all or part of the ultimate value of the enabling condition based on the states and/or values of some but not all of the attributes occurring in the enabling condition.

It is noted that the running time of this algorithm for executing the workflow S on an input is linear in the number of edges in D_s .

Given the above description and the pseudo code in Figs. 34A-34D, one skilled in the art could readily implement the algorithm.

The basic algorithm will therefore update the snapshot when a new attribute value is computed and the updated snapshot allows a determination to be made as to whether a task is eligible and/or unneeded. As described above, another characteristic of tasks, namely necessary, is also used in order to improve the performance of the decision

engine 104. If a task is known to be necessary to complete the workflow execution, then that task should be given high priority for evaluation.

Pseudo code for the extended algorithm for determining whether a task is necessary is described below in conjunction with Figs. 35A-35G. In order to describe the extended algorithm, several definitions must be introduced.

Given an extended snapshot $s = (\sigma, \mu, \alpha, \text{Hidden} - \text{att}, \text{Hidden} - \text{edge})$, an attribute A is a *relative source attribute* if each in-edge of A is an element of *Hidden-edge*.

A snapshot $s = (\sigma, \mu, \alpha, \text{Hidden} - \text{att}, \text{Hidden} - \text{edge})$ is *eager* if and only if:

- (a) For each relative source attribute A in S ,
 $\sigma(A) \in \{VALUE, DISABLED, READY + ENABLED\}$;
- (b) For each non-relative source attribute A , $\sigma(A) \geq ENABLED$ iff $\alpha(n)=T$ where n is the root node of the enabling condition of A ;
- (c) For each non-relative source attribute A , $\sigma(A) \geq DISABLED$ iff $\alpha(n)=F$ where n is the root node of the enabling condition of A ;
- (d) For each non-relative source attribute A , $\sigma(A) \geq READY$ iff for each input attribute B of A , $\sigma(B) \in \{VALUE, DISABLED\}$;
- (e) For each non-relative source attribute A , $\sigma(A) \in \{COMPUTED, VALUE\}$ iff $\mu(A)$ is defined;
- (f) for each condition node n , $\alpha(n)$ is defined accordance with the basic algorithm (section 3450) based on the value of σ and μ ;
- (g) Hidden-node is defined in accordance with the basic algorithm (section 3474) based on the value of σ and μ ;
- (h) Hidden-edge is defined in accordance with the basic algorithm (section 3476) based on the value of σ and μ .

It is noted that the snapshots produced by the basic algorithm will satisfy the above criteria for an eager snapshot.

There are four properties that are useful in determining necessary tasks: True-Necessary, False-Necessary, Value-Necessary, and Stable-Necessary. True-Necessary

and False-Necessary properties give information about the necessary relationship between an attribute and a predicate. Intuitively, an Attribute A is *True-Necessary* for a condition node p if in order for $\alpha(p)=T$ (in later snapshots), the value of A must be known. More formally:

- 5 Let D be a dependency graph, and let $s = (\sigma, \mu, \alpha, \text{Hidden-att}, \text{Hidden-edge})$, be an eager snapshot. An attribute A is *True-Necessary* for a condition node p , if the following is true:
 if for each eager snapshot $s' = (\sigma', \mu', \alpha', \text{Hidden-att}', \text{Hidden-edge}')$
 such that $s < s'$, and $\alpha'(p)=T$, then $\sigma'(A)$ is in $\{\text{VALUE}, \text{COMPUTED}\}$.

10

Intuitively, an Attribute A is *False-Necessary* for a condition node p if in order for $\alpha(p)=F$ (in later snapshots), the value of A must be known. More formally:

- 15 Let D be a dependency graph, and let $s = (\sigma, \mu, \alpha, \text{Hidden-att}, \text{Hidden-edge})$, be an eager snapshot. An attribute A is *False-Necessary* for a condition node p , if the following is true:
 if for each eager snapshot $s' = (\sigma', \mu', \alpha', \text{Hidden-att}', \text{Hidden-edge}')$
 such that $s < s'$, and $\alpha'(p)=F$, then $\sigma'(A)$ is in $\{\text{VALUE}, \text{COMPUTED}\}$.

15

Value-Necessary and Stable-Necessary properties give information about the necessary relationship between two attributes. Intuitively, an attribute A is *Value-Necessary* for an attribute B if the value of A must be known for later snapshots in which the state of B is COMPUTED or VALUE. More formally:

20

- 25 Let D be a dependency graph, and let $s = (\sigma, \mu, \alpha, \text{Hidden-att}, \text{Hidden-edge})$, be an eager snapshot. An attribute A is *Value-Necessary* for an attribute node B , if the following is true:
 if for any eager snapshot $s' = (\sigma', \mu', \alpha', \text{Hidden-att}', \text{Hidden-edge}')$
 such that $s < s'$, and $\sigma'(B)$ is in $\{\text{VALUE}, \text{COMPUTED}\}$, then $\sigma'(A)$ is in $\{\text{VALUE}, \text{COMPUTED}\}$.

25

- 30 Intuitively, an attribute A is *Stable-Necessary* for an attribute B if the value of A must be known for later snapshots in which the state of B is VALUE or DISABLED (i.e. B is stable). More formally:

30

- 35 Let D be a dependency graph, and let $s = (\sigma, \mu, \alpha, \text{Hidden-att}, \text{Hidden-edge})$, be an eager snapshot. An attribute A is *Stable-Necessary* for an attribute node B , if and only if for any eager snapshot $s' = (\sigma', \mu', \alpha', \text{Hidden-att}', \text{Hidden-edge}')$ such that $s < s'$, and $\sigma'(B)$ is in $\{\text{VALUE}, \text{DISABLED}\}$, then $\sigma'(A)$ is in $\{\text{VALUE}, \text{COMPUTED}\}$.

35

Thus, there are two ways for an attribute A to be *Stable-Necessary* for attribute B : 1) A is Value-Necessary for B (this implies that A is enabled) and A is False-Necessary for the Root of the enabling condition of B ; or 2) A is True-Necessary for the root of the enabling condition of B and A is False-Necessary for the root of the enabling condition of B .

- 5 Thus, given these properties, an attribute A is *necessary* if A is Stable-Necessary for some target attribute.

Given the above properties and definition of necessary, it is noted that the notion of necessary is defined in an absolute sense, and includes all attributes whose values will be necessary for constructing a terminal snapshot using the basic algorithm (or more
10 generally, execution based on the construction of a sequence of eager snapshots, each one extending the preceding one). However, the actual computation of all necessary attributes is not possible in practical situations, e.g., if the condition language includes integers with addition and multiplication. Even if the condition language is limited to include only atomic types with equality, computing all necessary attributes is intractable. Thus, a
15 subset of necessary attributes is determined in order to improve the performance of a workflow execution.

The extended algorithm for finding necessary attributes uses certain propagation rules in order to determine whether certain attributes are true-necessary, false-necessary, stable-necessary or value-necessary. Generally, the framework for the extended
20 algorithm is as follows. Given a dependency graph which is maintained as described above in connection with the basic algorithm, attributes will have certain states associated with them, and certain attributes may be hidden. For each attribute which is not hidden and which is in the state READY or READY+ENABLED, the algorithm will determine whether the attribute is true-necessary, false-necessary, stable-necessary or value-
25 necessary for some node. Using the propagation rules described below, these properties may be propagated up the dependency graph. If an attribute is found which is stable-necessary for a target attribute, then that attribute is considered necessary for completion of the workflow. These propagation rules will now be described.

First, a definition is necessary. A node n is a *relative-predecessor* of a node m in snapshot s if the edge (n,m) is an edge in dependency graph G and $(n,m) \notin \text{Hidden-edges}$. Given this definition, three sets of propagation conditions are now given. The
30

first set gives sufficient conditions to propagate False-necessary and True-necessary properties across condition nodes. The second set gives sufficient conditions for a node A to be stable-necessary. The last set gives a sufficient condition for a node A to be Value-necessary.

- 5 The sufficient propagation conditions for True-necessary and False-necessary are as follows.

Let s be an eager snapshot, A an attribute node, and p a non-hidden predicate node:

- 10 (1) When p is an OR node, then A is true-necessary for p , if A is true-necessary for all the relative predecessors of p .
- (2) When p is an OR node, then A is False-Necessary for p , if A is false-necessary for at least one direct predecessor of p .
- 15 (3) When p is an AND node, then A is true-necessary for p , if A is true-necessary for at least one relative predecessor of p .
- (4) When p is an AND node, then A is false-necessary for p , if A is False-necessary for all relative predecessors of p .
- 20 (5) When p is a NOT node, then A is true-necessary for p , if A is false-necessary for the relative predecessor of p .
- (6) When p is a NOT node, then A is False-necessary for p , if A is true-necessary for the relative predecessor of p .
- 25 (7) When p is a VAL(B) predicate, then A is true-necessary for p , if A is True-necessary for the root node of the enabling condition attached to B .

30

(8) When p is a $VAL(B)$ predicate, then A is false-necessary for p , if A is false-necessary for the root node of the enabling condition attached to B .

5 (9) When p is a $DIS(B)$ predicate, then A is true-necessary for p , if A is false-necessary for the root node of the enabling condition attached to B .

(10) When p is a $DIS(B)$ predicate, then A is false-necessary for p , if A is true-necessary for the root node of the enabling condition attached to B .

10 (11) When p is a predicate of the form $pred(t_1, \dots, t_k)$, then A is true-necessary for p , if it is value-necessary for at least one relative predecessor of p .

(12) When p is a predicate of the form $pred(t_1, \dots, t_k)$, then A is false-necessary for p , if it is value-necessary for at least one relative predecessor of p .

15

The sufficient propagation conditions for stable-necessary are as follows. Let s be an eager snapshot, and let A and B be attribute nodes where A and B are not hidden:

20 (13) A is Stable-necessary for B , if $\sigma(A) \geq ENABLED$ and B is enabled in s and A is Value-necessary for B .

(14) A is stable necessary for B if $\sigma(A) \geq ENABLED$ and B is not enabled in s and A is value-necessary for B and A is false necessary for the root of the enabling condition of B .

25

(15) A is stable necessary for B if $\sigma(A) \geq ENABLED$ and B is not enabled in s and A is true-necessary for the root for the enabling condition of B and A is false-necessary for the root of the enabling condition of B .

The sufficient propagation conditions for value-necessary are as follows. Let s be an eager snapshot, and let A and B be attribute nodes where A and B are not hidden:

(16) A is value-necessary for B if $\sigma(A) \geq \text{ENABLED}$ and A is an input attribute of B .

5

(17) A is value-necessary for B if $\sigma(A) \geq \text{ENABLED}$ and A is stable-necessary for at least one of the input attributes of B .

(18) A is value-necessary for A if $A \in \{\text{ENABLED}, \text{READY} + \text{ENABLED}\}$.

10

Finally, the propagation rule for an attribute to be necessary is as follows:

(19) An attribute A is necessary if it is stable-necessary for a target attribute.

The pseudo code for the extended algorithm for computing whether an attribute is necessary is shown in Figs. 35A - 35G. This extended algorithm is an extension of the basic algorithm described above in conjunction with Figs. 34A-34D. Thus, the extended algorithm could be considered the main algorithm, which in turn calls portions of the basic algorithm. The extended algorithm starts at the beginning of the execution of a workflow instance and ends when the execution is complete. At each step of the execution, it computes the necessary attributes and marks them by setting a corresponding element in an array to true. This algorithm is based on the 19 propagation rules described above. The basic approach of the algorithm is, for each attribute A , to propagate along the dependency graph the properties true-necessary, false-necessary, value-necessary, and stable necessary. When the property stable-necessary for an attribute node A reaches a target attribute node, this means that attribute A is necessary.

The extended algorithm takes advantage of the stability (i.e. once discovered, a necessary property cannot change) of the necessary properties to avoid re-computing necessary properties discovered in prior steps of the execution. The algorithm is incremental in the sense that it propagates the necessary properties along the dependency graph G by only computing new necessary properties at each step. The necessary properties are kept track of by four global variables, $F_N[]$, $T_N[]$, $V_N[]$, and

$S_N[][]$, each of which are integer matrices. $T_N[][]$ associates an integer value to each pair (p,A) where p is a condition node in G and A is an attribute node in G . $T_N[p][A]=0$ indicates that the attribute A is true-necessary for the condition node p . $F_N[][]$ associates an integer value to each pair (p,A) where p is a condition node in G and A is an attribute node in G . $F_N[p][A]=0$ indicates that the attribute A is false-necessary for the condition node p . $V_N[][]$ associates an integer value to each pair (B,A) where B and A are attribute nodes in G . $V_N[B][A]=0$ indicates that the attribute A is value-necessary for the attribute node B . $S_N[][]$ associates an integer value to each pair (B,A) where B and A are attribute nodes in G . $V_N[B][A]=0$ indicates that the attribute A is stable-necessary for the attribute node B .

At the beginning of the execution, each element of these matrices is initialized by a positive integer value. The initial value indicates how many decrements are required by the algorithm to guarantee that the corresponding necessary property is true (i.e. value=0). During execution, the algorithm decrements the value according to the propagation rules. For example, if p is an OR node, $T_N[p][A]$ is initialized by the number of incoming edges to p . This corresponds to rule number (1) which states that A is true-necessary if all its predecessors are true necessary. $F_N[p][A]$ is initialized with 1 since rule (2) states that A is false necessary as soon as one of its predecessors is false-necessary.

The extended algorithm will now be described in conjunction with Figs. 35A-G. The global variables are defined in section 3502. The variables defined in section 3504 are the same as those defined in section 3402 (Fig. 34) with respect to the basic algorithm. The remaining global variables in section 3502 have been described above. The initialization phase of the algorithm is shown in section 3506. This section initializes the variables. Section 3508 is a call to the initialization phase of the basic algorithm, which was described above in conjunction with section 3406 of Fig. 34. Section 3510 initializes the variables $T_N[][]$ and $F_N[][]$ as described above in accordance with the propagation rules. In section 3511, for each OR condition node p , the corresponding entries $T_N[p][A]$ are initialized to the number of predecessors of the node, in accordance with propagation rule (1). This is because for an attribute A to be true-necessary for p , A must be true necessary for all relative predecessors of p . In this way,

$T_N[p][A]$ will not reach 0 (indicating that attribute A is true necessary for condition node p), until it is decremented for each relative predecessor of p . Also, for each OR condition node p , the corresponding entries $F_N[p][A]$ are initialized to 1, in accordance with propagation rule (2). This is because for an attribute A to be false-necessary for p ,
 5 A must be false necessary for at least one relative predecessor of p . In this way, $F_N[p][A]$ will reach 0 (indicating that attribute A is false necessary for condition node p), when it is decremented for any one relative predecessor of p .

The remainder of section 3510 continues in a similar manner initializing T_N and F_N for AND nodes, NOT nodes, nodes of the form $VAL(A)$, $DIS(A)$, and $pred(t_1, \dots, t_k)$
 10 in accordance with the corresponding propagation rules. The corresponding propagation rules are indicated in section 3510 of Fig. 35B. Further, in section 3512 all attributes are initialized to not stable-necessary and not value-necessary by setting $S_N[][]$ to 1 and $V_N[][]$ to 1 respectively. In section 3513 all attributes are initialized to not necessary by setting $N[]$ to false.

15 The remaining portion of the extended algorithm is the incremental phase of the algorithm. This phase is called at each step of the workflow execution when a new attribute value is obtained as part of the operation of the prequalifier 2804. It updates the T_N , F_N , V_N , S_N , and N data structures according to the new eager snapshot computed by the basic algorithm. The incremental phase contains three steps. The first
 20 step is the preparation step. This step records information about the difference between the previous eager snapshot and the new eager snapshot computed by the basic algorithm. The second step is the instigation step which computes new necessary properties which immediately follow from the new information in the snapshot. The third step is the propagation step which propagates the new necessary properties computed in the
 25 instigation step.

The variables for the increment phase are defined in section 3516. Section 3518 defines the variables that are used to store the status of the current snapshot. These variables are described in further detail in the figure. Section 3520 defines variables that store attributes which are newly ENABLED or newly READY+ENABLED and edges
 30 which are newly hidden. Section 3522 defines variables which are used to store attributes which have become newly value-necessary (new_V_N), newly stable-

necessary (new_S_N), newly true-necessary (new_T_N), or newly false-necessary (new_F_N), as a result of the instigation step.

The preparation step is shown in section 3524. This step sets the values of prev_hidden_edges and prev_E and then calls the increment procedure of the basic algorithm in step 3526. The increment procedure is described above in conjunction with section 3414 of Fig. 34B. After execution of the increment procedure, there is a new snapshot which is now operated upon by the remainder of the extended algorithm.

The instigation step is shown in section 3528. This step is divided into 4 cases. Case 1 is shown in section 3530. This case implements propagation rule (18) which indicates that an attribute A is value-necessary for itself if it is in the state ENABLED or ENABLED + READY. This section 3530 thus determines which attributes A have newly entered the set of states $\{\text{ENABLED}, \text{ENABLED} + \text{READY}\}$ and for those attributes, sets $V_N[A][A]=0$, thus indicating that the attribute is value-necessary for itself. The pair (A,A) is also added to the set of pairs of attributes which are newly value-necessary (new_V_N). Case 2, shown in section 3532 implements propagation rule (13) which indicates that an attribute A is stable-necessary for an attribute B if the state of A is greater than or equal to ENABLED (i.e. ENABLED or ENABLED + READY), B is enabled, and A is value-necessary for B . This section 3532 applies this propagation rule for each newly enabled attribute B (i.e., those attributes in the set Δ_E) and updates new_S_N as appropriate.

As shown in section 3534, the variable Δ_HIDDEN_EDGE is used to hold edges that have been newly hidden during this iteration of the algorithm. Variables prev_T_N , prev_F_N , new_T_N and new_F_N are used to keep track of node-attribute pairs that become true-necessary or false-necessary during this execution of the algorithm.

Case 3, shown in section 3536 implements propagation rule (1) and operates as follows. If an edge (n,p) is hidden, then the predicate node n was computed to be false, in which case it is no longer relevant whether attribute A is true-necessary for n . Thus, if attribute A is not already true-necessary for n (i.e. $T_N[p][A] \neq 0$) then the value of $T_N[p][A]$ is decremented, which reduces the number of relative predecessors for which A needs to be true-necessary. Case 4, shown in section 3538 implements propagation rule (4) and operates as follows. If an edge (n,p) is hidden, then the predicate node n was

computed to be false, in which case it is no longer relevant whether attribute A is false-necessary for n . Thus, if attribute A is not already false-necessary for n (i.e. $F_N[p][A] \neq 0$) then the value of $F_N[p][A]$ is decremented, which reduces the number of relative predecessors for which A needs to be false-necessary.

5 The propagation step 3540 calls the `new_propagate` routine, which is shown in section 3542. The `new_propagate` routine receives, and operates on, the set of attributes which have been found to be newly value-necessary (`new_V_N`), stable-necessary (`new_S_N`), true-necessary (`new_T_N`), or false-necessary (`new_F_N`) as a result of the instigate step. Section 3544 calls the appropriate propagation routine for the newly
10 necessary attributes. Also, for attributes which are newly value-necessary, propagation rule 16 is implemented in section 3546.

 The newly value-necessary attributes are propagated in the `propagate_V_N` routine 3548. Section 3550 implements propagation rule (13). If the condition is satisfied, then A is set to stable-necessary for B (i.e. $S_N[B][A]$ is set to 0), and this new
15 necessary property is further propagated by calling the `propagate_S_N` routine. Propagation rule (14) is implemented in section 3552 and if an attribute is found to be stable-necessary as a result, that property is further propagated by calling the `propagate_S_N` routine. Propagation rule (11) is implemented in section 3554 and if an attribute is found to be true-necessary as a result, that property is further propagated by
20 calling the `propagate_T_N` routine. Propagation rule (12) is implemented in section 3556 and if an attribute is found to be false-necessary as a result, that property is further propagated by calling the `propagate_F_N` routine.

 The newly stable-necessary attributes are propagated in the `propagate_S_N` routine 3558. Propagation rule (17) is implemented in section 3560 and if an attribute is
25 found to be value-necessary as a result, that property is further propagated by calling the `propagate_V_N` routine. Propagation rule (19), which determines whether an attribute is necessary for the workflow execution, is implemented in step 3562

 The newly false-necessary attributes are propagated in the `propagate_F_N` routine 3564 and the newly true-necessary attributes are propagated in the `propagate_T_N`
30 routine 3566. The rules implemented by various portions of these routines are indicated

in Fig. 35. These routines would be well understood by one skilled in the art given the above description of the other propagation routines.

It is to be understood that the extended algorithm is only one implementation of the propagation rules described above. One skilled in the art could readily implement these propagation rules using other algorithms. Further, one skilled in the art could also use and implement other or additional propagation rules in accordance with the teachings of the present invention.

It is noted that the running time of the extended algorithm when executing workflow schemas S on one input is $O(|N||E|)$ where N is the set of nodes in D_s and E is the set of edges in D_s .

A task may be identified as necessary because the value produced by the task is value-necessary for some target attribute, i.e., the value produced by the task is used, either directly or indirectly, in the evaluation of the target attribute. A task may be identified as necessary because the value produced by the task is true-necessary and false-necessary for a target attribute, i.e., the value produced by the task is necessary, either directly or indirectly, to determine that the enabling condition for the target attribute is true and that it is necessary, either directly or indirectly, to determine that the enabling condition for the target attribute is false.

Given the above description and the pseudo code in Figs. 35A-35G, one skilled in the art could readily implement the algorithm.

Thus, the algorithms shown in Figs. 34 and 35 compute the three key properties of eligible, unneeded, and necessary. Referring now to Fig. 28, the algorithms of Figs. 34 and 35 are executed by the prequalifier 2804 and thus the three properties are computed by the prequalifier 2804. Tasks which are eligible may be provided to the candidate task pool 2802 for eager evaluation. However, if an eligible task is determined to be unneeded, then it is either not provide to, or removed from, the candidate task pool 2802. Further, if an eligible task is determined to be necessary, then it is marked as high priority in the candidate task pool 2802 so that it may be scheduled by the task scheduler 2806 for high priority execution by the execution engine 2812. This improves the performance of the overall operation of the decision engine 104. The prequalifier 2804 updates the candidate task pool 2802 after all source attributes have been processed, and also after a

new attribute value has been computed. In this manner, tasks which are known to be necessary for the completion of the workflow (eligible and necessary tasks) will be performed before tasks which are merely eligible. This is desirable because tasks which are merely eligible may actually be unneeded, and thus such tasks should not be given high priority.

It is not required that the enabling conditions of modules involve or refer to events, such as the initiation or completion of tasks, i.e., the executions of modules. In the context of DL specifications, conditions may test only the stable states and values of attributes and modules. Thus, there is an implicit dependence between the truth value of an enabling condition and the times at which the modules and attributes referred to in the condition become stable. However, once these modules and attributes become stable they cannot change value, and so the truth value of the condition will remain the same for the duration of the execution of the workflow instance. This is a result of the acyclicity condition imposed on DL specifications and the fact that each attribute is produced by only one module. Thus, once the truth value of an enabling condition is established, the particular times at which that truth value is tested by an execution algorithm will not affect the overall outcome of the workflow instance. In particular, unless the enabling conditions explicitly refer to the timing of module execution, the duration of processing of tasks will not affect the truth value of an enabling condition, and, in the absence of optimizations as described above, will not affect whether or not a given module is executed during a workflow instance.

The independence of module execution in workflows based on DL specifications stands in marked contrast with workflow systems that use enabling conditions that are required to explicitly refer to events such as the initiation or completion of tasks.

Enabling conditions in such systems have the form “on <event> if <condition>”. The intended semantics is that during execution the <condition> should be tested immediately after the <event> occurs. In these systems, the truth value of <condition> may be defined and change value over time. Thus, the outcome of testing the enabling condition, i.e., the decision of whether the corresponding module is executed or not, may depend on the exact time that the <event> in the enabling condition occurs. In particular, the enabling conditions and the decisions they embody may depend on the durations of execution of

different modules. This dependence implies that analysis of the behavior of such systems is at roughly the same level of difficulty as the analysis of the behavior of procedural programs.

As described above, decision modules are evaluated using computation rules and a combining policy. In addition, a novel graphical user interface (GUI) is used to display a representation of the evaluation of decision modules. The GUI is particularly advantageous for understanding and debugging the semantics of the workflow system, and for understanding how different execution strategies affect the processing of different kinds of inputs.

In describing the GUI, we again make the simplifying assumptions made above that we are given a declarative module in which each internal module produces exactly one output attribute and may have a side-effect. Further, once enabled, it is assumed that all internal modules will always produce a value and will never produce an exception value. For this discussion, the term *non-decision module* refers to internal modules that are not decision modules. The term *non-decision attribute* refers to an attribute whose defining module is a non-decision module.

The GUI may be implemented in connection with essentially any policy for evaluating decision attributes, i.e., those attributes that are evaluated as specified by a decision module. In order to illustrate the GUI most clearly, we do not use the policy for evaluating decision attributes described above. Instead, the GUI is described using an execution policy that is eager with respect to the evaluation of computation rule conditions and computation rule terms. The contribution rule terms are also referred to as "contributions" because, as described above, these terms contribute values to an attribute if the condition is true. Given the description herein, one skilled in the art could readily implement the GUI in connection with other execution policies.

In order to describe the eager execution policy for decision attributes, we modify the notion of snapshot used earlier, in two ways. The first modification is to restrict the set of states that decision modules can be in, and the second modification is to permit computation rule conditions and contributions to be evaluated in an eager fashion.

Recall in the previous discussion that non-side effect modules can have states as shown in Figure 29. In the current discussion we use a refinement of the state diagram

shown there for decision modules. Specifically, we use the FSA of Fig. 36 for decision modules. Each decision module starts in the state READY, even if the source attributes for the decision module are not yet stable. The conditions and/or contributions of rules in a decision module may be evaluated eagerly, as the attributes used by those conditions and/or contributions become stable. If all the rule conditions are evaluated, and the contributions of all rules with true condition are evaluated, then the value for the attribute can be determined, and the module is moved into state COMPUTED. Alternatively, if the enabling condition of the decision module is determined to be true, the module may move to state READY+ENABLED. The states VALUE and DISABLED are the same as described above in connection with Fig. 29.

Fig. 37 shows the FSA used for each computation rule. Each rule begins in state READY. If a sufficient number of attributes in the rule condition become stable for a determination to be made that the rule condition is, or will eventually become, TRUE, then the rule moves to state CONDITION_TRUE. Alternatively, if a sufficient number of attributes in the rule contribution become stable such that the eventual value of the contribution can be computed, then the rule moves to state CONTRIBUTION_COMPUTED. Thereafter, if both the rule condition is determined to be true and the contributed value is computed, then the rule moves to state CONTRIBUTED_VALUE. Alternatively, if a sufficient number of attributes in the rule condition become stable for a determination to be made that the rule condition is, or will eventually become, FALSE, then the rule moves to state CONDITION_FALSE.

Further in connection with the description of the GUI, the notions of workflow schema and snapshot presented above are modified. First, assume that a workflow schema has the form $S = (Att, Src, Tgt, Eff, Cnd, Mod, Dec)$, where

1. Components *Att*, *Src*, *Tgt*, *Eff*, *Cnd*, and *Mod* are as described above; and
2. *Dec* is a set of pairs $\{(Rules_A, CP_A) \mid A \text{ is a decision attribute}\}$, where for each decision attribute *A*, *Rules_A* is the set of computation rules in the module outputting *A*, and *CP_A* is the combining policy for the module outputting *A*.

For schema *S*, we use *Rules* to denote the set $\cup \{ Rules_A \mid A \text{ is a decision attribute in } S \}$, i.e., the set of all computation rules occurring in the decision modules of *S*.

Second a *snapshot* for S is defined to be a pair $s = (\sigma, \mu)$ where

1. the *state mapping* σ is a total function from $Att \cup Rules$ such that σ maps
 - a. each decision module to $\{READY, ENABLED+READY, COMPUTED, VALUE, DISABLED\}$,
 - b. each non-decision, non-side-effect module to $\{UNINITIALIZED, ENABLED, READY, ENABLED+READY, COMPUTED, VALUE, DISABLED\}$,
 - c. each non-decision, side-effect module to $\{UNINITIALIZED, ENABLED, READY, ENABLED+READY, VALUE, DISABLED\}$, and
 - d. each computation rule to $\{READY, CONDITION_TRUE, CONTRIBUTION_COMPUTED, CONTRIBUTED_VALUE, CONDITION_FALSE\}$.
2. The *value mapping* μ is a partial function from $Att \cup Rules$ such that μ maps
 - a. Att into $\cup \{\tau(A) \mid A \in Att\}$, such that for each A , if $\mu(A)$ is defined then $\mu(A) \in \tau(A)$, and such that for each A , $\mu(A)$ is defined iff $\sigma(A) = VALUE$ or $\sigma(A) = COMPUTED$.
 - b. each rule r in $Rules$ to a value with the type of the contribution of r , such that $\mu(r)$ is defined iff $\sigma(r) = CONTRIBUTION_COMPUTED$ or $\sigma(r) = CONTRIBUTED_VALUE$.

One snapshot s' *extends* another snapshot s (specified as $s < s'$) if for each attribute A :

- (a) if A has a value in s , then A has the same value in s' ; and
 - (b) the state of A in $s' \geq$ the state of A in s , where \geq is relative to the FSAs of Figs. 29, 30 and 36
- and for each computation rule r :
- (c) if r has a contributed value in s , then r has the same value in s' ; and
 - (d) the state of r in $s' \geq$ the state of r in s , where \geq is relative to the FSA of Fig. 37.

Snapshot s' *strictly extends* snapshot s if $s < s'$ and $s \neq s'$. A snapshot is *complete* if each attribute is stable and each computation rule is in state $CONTRIBUTED_VALUE$ or $CONDITION_FALSE$. A snapshot is *terminal* if each target attribute is stable.

For the purposes of describing the GUI, it is not necessary to use a specific algorithm or policy for executing a declarative workflow. We assume here that execution of workflow *S* begins with a snapshot such that all source attributes are stable, all internal modules are in state UNINITIALIZED (or READY, for decision modules), and all

5 computation rules are in state READY. Then a sequence of snapshots is constructed, each one a strict extension of the previous one. Execution halts when a terminal snapshot is reached.

The GUI will now be described in connection with the declarative module INFO_ABOUT_CUSTOMER as shown in Figure 5. This module has two source

10 attributes, CUST_REC and ACCOUNT_NUMBER. There are eight internal modules 504, 508, 512, 516, 520, 524, 528, and 532. Modules 504, 508, and 512 are non-decision modules. Modules 516, 520, 524, 528, and 532 are decision modules. The specification of the INFO_ABOUT-CUSTOMER module and its internal modules has been described in detail above. Reference to that description may be helpful during the following

15 description of the GUI.

Figs. 38 and 39 are illustrative display screen shots and are used to illustrate the GUI. The figures show information about two snapshots that might arise during a hypothetical execution of the INFO_ABOUT_CUSTOMER module. Fig. 38 shows execution information near the beginning of execution and Fig. 39 shows execution

20 information somewhere in the middle of execution.

Referring to Fig. 38, the display is in a grid format, with the rows labeled with numbers and the columns labeled with letters. The intersection of a row and a column defines a cell. Each column of the display corresponds to an attribute of the INFO_ABOUT_CUSTOMER module. The first two rows provide information about

25 how the attributes are computed. Row 1 indicates the name of the module computing the attribute. For ease of cross-reference, row 1 of Fig. 38 includes the corresponding call-out identification of the module from Fig. 5. Such call-out numbers would not be included in an actual embodiment of the GUI. Row 2 indicates the manner of computation. For non-decision modules, row 2 indicates the type of the module (e.g.,

30 "foreign"). For decision modules, row 2 displays a short description of the combining policy of the module. This short description could be specified, for example, in the

textual description of the combining policy. As an alternative, row 2 could display the name of the function specifying the combining policy for the module. The attribute names are shown in row 3.

Rows 4 and 5 display the evaluation status of attributes. The fourth row displays the value of an attribute if the attribute is stable with an assigned value. For example, in Fig. 38, the two source attributes corresponding to columns A and B, are stable with values. In particular, the CUST_REC attribute has as a value a tuple with first fields being name = "John Doe", address = "101 Ash, LA", card_color = "gold", and hates_promos? = FALSE. The ACCOUNT_NUMBER attribute has a value of 421135.

The cells representing these attribute values also display the label "SV" indicating that the attributes are stable with an assigned value. The remaining cells in row 4 display the label "NS", indicating that the corresponding attributes are not stable. The fifth row of the display displays the states of the modules. The three foreign modules (shown in columns C, D, and E) are in state ENABLED+READY, a consequence of the fact that their enabling conditions are all TRUE and their input attributes are stable. The module CALCULATE_CUST_VALUE is also in state ENABLED+READY as shown in cell I5. This is because its associated enabling condition is TRUE, and by assumption all decision modules begin in state READY. The other decision modules are in state READY, because the attributes used in their enabling conditions are not yet stable.

Rows 6, 7, 8, ... are used to indicate the evaluation status of computation rules. Accordingly, cells are shown in these rows only for decision modules corresponding to columns F, G, H, I, and J. The cells in rows 6, 7, 8, ... are called *rule cells*. For each attribute *A* whose value is computed by a decision module, there is a one-to-one correspondence between the computation rules for *A* and the rule cells in the column for *A*. Note that for clarity only the first 5 rule cells for attribute NET_PROFIT_SCORE (column G) are shown, even though Figure 16 shows this attribute as having 7 computation rules.

The evaluation status of the computation rules at a point in the execution is indicated in the corresponding rule cells. The states of READY and CONDITION_TRUE are indicated by labels within the cell. The states of CONTRIBUTION_COMPUTED and CONTRIBUTED_VALUE are indicated by

placing a value in the cell along with the label C-V to indicate a state of CONTRIBUTED_VALUE or the label C-C to indicate a state of CONTRIBUTION_COMPUTED. The state of CONDITION_FALSE is indicated by placing the symbol \perp , representing a null value, in the cell.

5 In the embodiment described here, it is assumed that computation rule conditions and contributions are evaluated eagerly. In Fig. 38, cells G9, H6, H8, I7, and I9 indicate that the corresponding rules are in state CONDITION_FALSE. All of these rules have conditions based on the card color of the customer, which is known from the value of attribute CUST_REC. Similarly, cell H7 is in state CONDITION_TRUE because the
 10 condition for the corresponding rule is CUST_REC.card_color = "gold". However, the contribution for the rule corresponding to cell H7 depends on the ACCOUNT_HISTORY attribute, which is not yet stable as indicated by cell 25. In contrast, cells G10 and I8 are in state CONTRIBUTED_VALUE, because their corresponding rule conditions are true and the rule contributions depend on no attributes (and hence, on no attributes that are
 15 currently unstable). Cell J6 is in state CONTRIBUTION_COMPUTED because the corresponding rule condition depends on a non-stable attribute as indicated by cell J4, but the rule contribution is the constant value "collect". The remaining rule cells are in state READY, since both their rule conditions and contributions depend on attributes that are currently not stable.

20 Fig. 39 shows an example display screen shot after several steps have occurred in the execution of the workflow and the evaluation of the attributes has progressed. In particular, Fig. 39 shows that the attributes RECENT_PURCHASES and ACCOUNT_HISTORY have returned values as shown in cells 25 and E4 respectively. A value for attribute RECENT_CONTACTS has not yet been received as indicated by
 25 cell 15. Based on this partial information, it has been determined in the execution that the CALCULATE_NET_PROFIT_SCORE module is disabled as indicated by the label DISABLED in cell G5. The associated attribute value cell, G4, now contains the null symbol \perp and the label "SU" indicating that the attribute value is stable and undefined.

Note that values for the conditions and contributions of two additional
 30 computation rules of the NET_PROFIT_SCORE attribute have been obtained during the execution that led from the display of Fig. 38 to the display of Fig. 39. Specifically, the

rule represented by cell G6 has become `CONDITION_FALSE` as indicated by the symbol \perp , and the rule represented by cell G8 has become `CONTRIBUTED_VALUE` as indicated by the value -9 and label C-V. Since the `NET_PROFIT_SCORE` attribute has become `DISABLED`, no further information about the computation rules shown in column G need to be computed, since the attribute will not contribute to the final outcome of the workflow execution.

The execution progression has also permitted evaluation of the computation rule corresponding to cell H7, and hence the evaluation of the `LATE_PAYMENTS_SCORE` attribute. The execution progression has also permitted evaluation of the condition of the rule corresponding to cell I6.

The algorithm for maintaining and dynamically updating the GUI display as described above is shown in Figs. 40A and 40B. The algorithm contains two main sections. The Initialization section is used to initialize the display prior to beginning execution of the workflow. The Iteration section is executed when new information is received from the execution engine 2812 and the display is to be updated with the new information.

We now describe one way that the processing for supporting the GUI could be incorporated into the basic algorithm of Figure 34. Because this algorithm views the execution of decision modules as external “black boxes”, the illustration here does not include a display of the incremental evaluation of computation rules. The Initialization step of Fig. 40A could be included at the end of part 3406 of Fig. 34A. The Iteration phase of Figs. 40A and 40B could be included in section 3414 of Figure 34B, just after section 3420. In this case, the Iteration phase would be applied multiple times, once for each relevant event that occurs during execution of section 3422. Alternatively, the Iteration phase of Figs. 40A and 40B could be included (a) into section 3414 of Fig. 34B just after section 3418 and (b) into section 3422 of Figs. 34B and 34C just after each occurrence of a command that assigns a state value to an attribute (i.e., a value for $\sigma[C]$ for some attribute C). Based on this description it would be clear to one skilled in the art how to incorporate processing to support the GUI into the extended algorithm of Fig. 35, and into algorithms that support execution of DL specifications that are eager with respect to the evaluation of computation rule conditions and/or computation rule terms.

As used throughout the description of the algorithm, various "indications" are applied to cells. An indication may be any type of visible indication, such as color, shading, pattern, outline, icon, or alphanumeric label, which conveys information to a user. Alphanumeric labels are used for the example screen shots shown in Figs. 38 and 40. Turning now to the algorithm of Figs. 40A and 40B, in line 4002 rows 1, 2, and 3 of the display are generated based on the DL specification. In section 4004 row 4 of the source attributes is initialized. If the source attribute has a value, then the value is inserted in the cell and the attribute_value_indication is applied to the cell. If the source attribute is disabled, then the attribute_disabled_indication is applied to the cell. In section 4006 the cells representing the non-decision modules are initialized by applying a module_uninitialized_indication in row 5 and an attribute_uninitialized_indication in row 4. In section 4008 the cells representing the decision modules are initialized by applying a module_ready_indication in row 5 and an attribute_uninitialized_indication in row 4. In section 4010 the rule cells are initialized by applying a rule_ready_indication to the rule cells in rows 6, 7, 8,

The Iteration section of the algorithm is now described. This section is one case statement such that the processing to be performed depends on the type of event received from the execution engine 2812. If the event is a non-decision module entering state ENABLED, then in section 4012 a module_enabled_indication is applied to the appropriate cell in row 5 of the display. If the event is a non-decision module entering state READY, then in section 4014 a module_ready_indication is applied to the appropriate cell in row 5 of the display. If the event is a non-decision module entering state READY+ENABLED, then in section 4016 a module_ready+enabled_indication is applied to the appropriate cell in row 5 of the display. If the event is a non-decision module entering state COMPUTED, then in section 4018 a module_computed_indication is applied to the appropriate cell in row 5 of the display, the computed value is displayed in the appropriate cell in row 4 of the display, and an attribute_computed_indication is applied to the cell in row 4. If the event is a non-decision module entering state VALUE, then in section 4020 a module_value_indication is applied to the appropriate cell in row 5 of the display, the cell in row 5 is labeled as "value", the assigned value is displayed in the appropriate cell in row 4 of the display, and an attribute_value_indication is applied

to the cell in row 4. If the event is a non-decision module entering state DISABLED, then in section 4022 a module_disabled_indication is applied to the appropriate cell in row 5 of the display, the cell in row 5 is labeled as "disabled", the \perp symbol is displayed in the appropriate cell in row 4 of the display, and an attribute_disabled_indication is applied to the cell in row 4. If the event is a decision module entering state ENABLED+READY, then in section 4024 a module_enabled+ready_indication is applied to the appropriate cell in row 5 of the display and the cell is labeled as "enabled+ready". If the event is a decision module entering state COMPUTED, then in section 4026 a module_computed_indication is applied to the appropriate cell in row 5 of the display, the cell in row 5 is labeled as "computed", the computed value is displayed in the appropriate cell in row 4 of the display, and an attribute_computed_indication is applied to the cell in row 4. If the event is a decision module entering state VALUE, then in section 4028 a module_value_indication is applied to the appropriate cell in row 5 of the display, the cell in row 5 is labeled as "value", the computed value is displayed in the appropriate cell in row 4 of the display, and an attribute_value_indication is applied to the cell in row 4. If the event is a decision module entering state DISABLED, then in section 4030 a module_disabled_indication is applied to the appropriate cell in row 5 of the display, the cell in row 5 is labeled as "disabled", the \perp symbol is displayed in the appropriate cell in row 4 of the display, and an attribute_disabled_indication is applied to the cell in row 4.

If the event is a computation rule entering state CONDITION-TRUE, then in step 4032 a rule_cond_true_indication is applied to the appropriate rule cell. If the event is a computation rule entering state CONTRIBUTION-COMPUTED, then in step 4034 the computed value is displayed in the appropriate rule cell and a rule_contribution_computed_indication is applied to the cell. If the event is a computation rule entering state CONTRIBUTED-VALUE, then in step 4036 the computed value is displayed in the appropriate rule cell and a rule_contributed_value_indication is applied to the cell. If the event is a computation rule entering state CONDITION-FALSE, then in step 4038 the \perp symbol is displayed in the appropriate rule cell and a rule_condition_false_indication is applied to the cell.

The above description described one embodiment of the GUI. Those skilled in the art could implement many variations of the GUI. Examples of such variations follows.

- 5 1. Different coloring and labeling conventions: Use different colors and/or patterns for indicating the state and/or other information about computation rules at different points in the execution. Use additional or different information in the labels for cells, e.g., include a time-stamp for when the value for a cell has been computed.
- 10 2. Different execution algorithms: show the progress of executions in conjunction with different algorithms for executing decision modules.
3. Different FSAs: The GUI can be used with FSAs for computation rules which are different than the FSA shown in Fig. 37.
- 15 4. User control over visual layout: Permit the user to hide or expose selected columns or rows of the display. Also, permit the user to click on cells to display more information about them. For example, clicking on a rule cell could result in the display of a pop-up window showing the rule. Clicking on a cell in row 2 could result in the display of the CPL program specifying the combining policy associated with that cell.
- 20 5. Different visual layout: In Figs. 38 and 39, attributes are positioned along the horizontal axis and rules positioned along the vertical axis. Many alternatives are possible. Some representative examples include: (a) position attributes along the vertical axis and rules along the horizontal axis; (b) instead of using a grid paradigm, show decision modules as hexagons as in Fig. 5 and show rule status for a given
- 25 attribute using a column or row of cells; (c) same as (b) but display the cells for rules in a tree-based or other structure that reflects the kinds of contributions different rules might make.
6. Use in conjunction with systems not based on DL specifications, including systems specified using, for example, flowcharts, procedural languages, or scripting
- 30 languages.
7. Batch display: The example described above illustrates how to display the execution of a single workflow instance. The visual paradigm can be used to display the result

of executions of a set of workflow instances. For example, the color of a rule cell might be based on the percentage of executions for which the condition of the corresponding rule was true. Rule cells might be labeled with an aggregate value (e.g., an average) indicating the family of contributions made by the rule in different executions.

8. Permit backtracking: The example assumed that the sequence of displays produced corresponded to an actual or hypothetical execution of the workflow. It is also possible to permit a user to halt the execution, and modify it by replacing the values of source or non-decision attributes.

9. Highlight data dependencies between cells: For example, the interface could permit the user to click on rule cells in order to display relationships between attributes and rules, e.g., what attributes does a rule condition depend on, or what attributes does a rule contribution depend on.

10. Incorporate general modules: The example assumed that each module produced exactly one output attribute. The GUI can be used in contexts where modules produce more than one attribute. This could be accomplished, for example, by permitting cells in the first, second and fifth rows to span a number of columns equaling the number of output attributes of a given module. (This was done for the source attributes, in columns A, B of row 1 in Figs. 38 and 39).

The foregoing Detailed Description is to be understood as being in every respect illustrative and exemplary, but not restrictive, and the scope of the invention disclosed herein is not to be determined from the Detailed Description, but rather from the claims as interpreted according to the full breadth permitted by the patent laws. It is to be understood that the embodiments shown and described herein are only illustrative of the principles of the present invention and that various modifications may be implemented by those skilled in the art without departing from the scope and spirit of the invention.